Self-stabilizing algorithm for checkpointing in a distributed system

Partha Sarathi Mandal, Krishnendu Mukhopadhyaya*

Advanced Computing and Microelectronics Unit, Indian Statistical Institute, 203 B. T. Road, Kolkata 700 108, India

Abstract

If the variables used for a checkpointing algorithm have data faults, the existing checkpointing and recovery algorithms may fail. In this paper, self-stabilizing data fault detecting and correcting, checkpointing, and recovery algorithms are proposed in a ring topology. The proposed data fault detection and correction algorithms can handle data faults; at most one per process, but in any number of processes. The proposed checkpointing algorithm can deal with concurrent multiple initiations of checkpointing and data faults. A process can recover from a fault, using the proposed recovery algorithm in spite of multiple data faults present in the system. All the proposed algorithms converge in $O(n)$ steps, where n is the number of processes. The algorithm can be extended to work for general topologies too.

Keywords: Data fault; Process fault; Checkpointing; Rollback recovery; Self-stabilization

1. Introduction

Checkpointing is a well-known technique for fault tolerance in distributed computing systems. 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 checkpoint saved in the non-volatile stable storage. This strategy usually works well in uniprocessor systems. The reconstruction of a distributed system with multiple processes is, however, not easy. Here, the system has to rollback to a *consistent global state*. A set of checkpoints, with one checkpoint for every process, is said to be a *consistent global checkpointing state* (*CGS*) if there does not exist any message such that the sender has no record of the sending but the receiver has the record of the receipt and the receiver has no record of the receiving but the sender has the record of the sending. The first type of message is called *orphan message* and the second type of message is called *missing message*. The missing messages may be acceptable, if messages are logged [1,10,17] by sender.

A self-stabilizing distributed system ensures recovery from an *illegitimate* configuration in a finite number of steps. A system may reach an *illegitimate* configuration due to failure or a perturbation in the system. Dijkstra [5] first introduced the notion of self-stabilization in distributed systems in 1973. His definition of self-stabilization was "regardless of its initial state, it is guaranteed to arrive at a legitimate state in a finite number of steps". Schneider [16] gave it a more formal representation. For a system S, a global state of the system is the union of the local states of its components. The predicate P identifies its correct execution. In other words, if the value of P is *False*, we note that the system is in an illegitimate state, otherwise the system is legitimate. "S is self-stabilizing with respect to predicate P, if it satisfies the following two properties:

Closure—P is closed under the execution of S. That is, once P is established in S, it cannot be falsified.

Convergence—Starting from an arbitrary global state, S is guaranteed to reach a global state satisfying P within a finite number of state transition" [16].

In this paper, we propose self-stabilizing checkpointing and recovery algorithms for unreliable ring topology which can detect and correct data faults. Two types of faults, *data fault* and *process fault*, are considered.

Data fault: It means that the data of a variable used by the checkpointing algorithm is changed or corrupted due to some unreliability of the system.

Process fault: It signifies process crash. Such a process loses the data stored in volatile storage and has to be rolled back. The process state is reconstructed from a checkpoint saved in the non-volatile stable storage.

Herman [9] compiled an extensive list of works on self-stabilization. Ghosh and He [8] proposed a scalable time-independent self-stabilization algorithm to stabilize from any k-fault configuration in a distributed system consisting of n processes on a tree topology. The worst case stabilization time and the space complexity of the method are $O(k^2)$ and $O(\Delta.(\Delta.k + \log_2 n))$ per process (where Δ is the maximum degree of a node), respectively. Ghosh et al. [7] proposed several ways of measuring the performances of fault-containing self-stabilizing algorithms. Several earlier works [2–4] on snapshot collection algorithms assume that at any point of time only one snapshot collection process is active. Koo and Toueg [11], Spezialetti and Kearns [18], Prakash and Singhal [15], Mandal and Mukhopadhyaya [12] have proposed methods for handling concurrent initiations of snapshot collection. Logical checkpoints were introduced by Wang et al. [22,23]. Vaidya presented three approaches [20] for taking a logical checkpoint. Vaidya used a simple approach [21] where physical checkpoints are arbitrarily staggered, but consistency is enforced by logging the messages during confirmation. Many recovery algorithms on distributed computing system have been proposed in the literature [6,14,17,19]. Manivannan and Singhal [14] proposed an asynchronous recovery algorithm in which all processes recover from their last existing checkpoints.

2. The underlying 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. While the synchronous checkpointing is in progress, a process sends checkpointing request (*ckpt_req*) along the anti-clockwise direction. However, application messages may travel in any direction. There is no common clock, shared memory or central coordinator. Message passing is the only mode of communication between any pair of processes.

A process, initiates checkpointing by taking a temporary *logical checkpoint* [20–23]. It increments its *v_no* by one and stores the new checkpoint in its non-volatile stable storage. We assume that the checkpoints stored in the stable storage cannot be corrupted. We also assume that the checkpointing state (*ckpt_state*) and checkpointing sequence number/version number (*v_no*), maintained in the main memory, might be corrupted or changed. If a process fails when a data fault is present (in any process in the system), the algorithm proposed in [3,12,13] may not give a CGS after rollback. Each process maintains a counter, called *v_no*. Each process may store at most two checkpoints, one permanent ($ckpt\_state = P$) and one temporary ($ckpt\_state = T$) when checkpointing algorithm is running or communication-induced [13] checkpoint is taken. Otherwise, a process will have a single permanent checkpoint. Two checkpoints will have two consecutive *v_no*s and they will belong to two distinct CGS. Interval between checkpoints with $v\_no = p$ and $p + 1$ is called pth *checkpoint interval*. Each process maintains a list of messages sent by it, in a *message logging table* (*MLT*). Whenever a process takes a checkpoint, it stores its *MLT* in stable storage along with the checkpoint. After receiving a message, the receiver sends an acknowledgement (*ack_msg*) with the receiving checkpoint interval (*receive_interval*). *receive_interval*s of the corresponding messages are also maintained in the *MLT*s by the senders. A message is deleted from an *MLT* only when a permanent checkpoint is taken by the receiver after processing the message, i.e., when the current checkpoint interval of the sender is greater than or equal to $receive\_interval + 2$.

At most one data fault per process is assumed. That fault may occur at any time during the computation. However, several processes in the system may have data faults at the same time. In the worst case, all processes can have data faults concurrently. The system can recover if a process fails even when a data fault is present in the system. The proposed scheme can handle data fault and process fault concurrently. The system is said to be in a *legitimate* configuration if there is no data fault and process fault, and there exists a *CGS* for the system. If at least one data fault or process fault is present in the system then the system reaches an *illegitimate* configuration. When a process detects data fault and process fault, it knows that it has reached an *illegitimate* configuration. In this situation, in case of data fault the process first tries to correct the data fault itself. If the process cannot do so all by itself, it communicates with other processes and recovers from the data fault. It may be noted that, the communication for correcting the fault does not generate extra messages. The required information is gathered by piggybacking some extra information on application messages or control messages. In case of process fault, system recovers by rolling back to the CGS and resumes computation. For all the above cases in a finite number of steps, the system reaches a *legitimate* configuration from an *illegitimate* configuration.

3. Motivation

If the variables used for a checkpointing algorithm [3,12,13] have data faults, the algorithm may fail. For example, in communication-induced checkpointing [13], if the variable representing the version number of the latest checkpoint in a process, gets changed due to a fault, a process may not take a checkpoint when it ought to. Similarly, in some other schemes too, if the version number is changed, finding a CGS may not be possible.

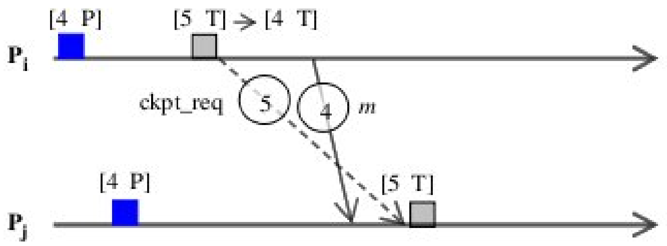


Fig. 1. An example showing inconsistency in checkpointing and recovery algorithms due to data fault.

Fig. 1 shows an example how communication-induced checkpointing algorithms and rollback recovery algorithms fail due to a data fault in checkpoint *v_no*. The order pair [*v_no*, *ckpt_state*] in the figure represents two checkpointing variables corresponding to a checkpoint. Process P_i initiates checkpointing by taking a temporary checkpoint with $v\_no = 5$ and sends *ckpt_req* to P_j. At this point of time, the *v_no* of P_i gets changed from 5 to 4 due to some unreliability of the system. Then, P_i sends an application message m to P_j, piggybacking the value of the *v_no* $(= 4)$ of the latest checkpoint. On receiving m, P_j observes that the *v_no* $(= 4)$ tagged with m is not greater than the latest checkpoint $v\_no = 4$ of P_j. P_j receives *ckpt_req* after processing m and takes temporary checkpoint with $v\_no = 5$. At this point if a process fails and the system recovers by rolling back to the latest checkpoints of P_i and P_j; system does not reach a CGS. Message m is an orphan message.

4. Predicates for self-stabilization

Process, P_i $\forall i \in \{0, 1, \ldots, n-1\}$ maintains the following four variables in their volatile storage. Two values of the variables *v_no* and *ckpt_state* (P/T) are associated with a checkpoint:

$prev_i$	: *v_no* of the previous checkpoint
$state\_prev_i$	: *ckpt_state* of the previous checkpoint
$curr_i$	: *v_no* of the current checkpoint
$state\_curr_i$	: *ckpt_state* of the current checkpoint

Each process maintains two predicates:

$pred_1$	: returns *True* when the values of version numbers (*v_no*) of two checkpoints are consecutive
$pred_2$	: returns *True* when *ckpt_state* of the previous checkpoint is permanent (P)

$pred_1$ and $pred_2$ are computed as follows:

if $(curr_i = prev_i + 1)$ **then** $pred_1 = True$ **else** $pred_1 = False$ **end if**,

if $(state\_prev_i = P)$ **then** $pred_2 = True$ **else** $pred_2 = False$ **end if**.

If a process is in a legitimate state, both $pred_1$ and $pred_2$ should return values *True*. If one of the predicates returns value *False*, the process is in an illegitimate state. We do not consider the case where a single process may have more than one error.

When a data fault is detected, if possible, the process corrects itself; otherwise it takes help from other processes. A process will check its predicates whenever it sends an application message, control message or an application message is passing through the process with an *undecided* information.

5. Data fault detection and correction

Process, P_i checks its predicates before sending an application message then logs the message in the MLT. If $pred_2$ returns *False*, P_i corrects the fault by putting $state\_prev_i = P$. Since at most one data fault in a process is assumed, $pred_1$ returning *False* implies that the fault is either in $curr_i$ or in $prev_i$. If $prev_i$ is faulty, then the correct value for $prev_i$ would be $curr_i - 1$. If $curr_i$ is faulty then the correct value for $curr_i$ would be $prev_i + 1$. In such a case, P_i cannot decide which one would be correct. P_i sends an *undecided* (U) tag with the application message. If $pred_1$ returns *True*, P_i sends the application message with tag D. P_i sends an application message to the next process with i (sender id), k (receiver id), $prev_i$, $state\_prev_i$, $curr_i$, $state\_curr_i$.

When P_j receives a message with tag U, if $pred_1$ is *True*, then P_j corrects the fault of sender (P_i) of this message. If $pred_1$ is *False*, and if one of the following condition is *True*, then P_j would not be able to correct the fault of P_i and its own. Now, P_j also become undecided.

Condition 1 $((prev_i = prev_j) \wedge (curr_i = curr_j) \wedge (state\_curr_i = state\_curr_j))$

Condition 2 $((prev_i = prev_j + 1) \wedge (curr_i = curr_j + 1) \wedge (state\_curr_i = T) \wedge (state\_curr_j = P))$

Condition 3 $((prev_j = prev_i + 1) \wedge (curr_j = curr_i + 1) \wedge (state\_curr_j = T) \wedge (state\_curr_i = P))$

If none of the above three conditions is *True*, P_j corrects the fault. Given that $pred_1$ is *False* for P_i, $curr_i \neq prev_i + 1$. Let $S_i^1 = (curr_i - 1, curr_i)$, and $S_i^2 = (prev_i, prev_i + 1)$. The correct value for the ordered pair $(prev_i, curr_i)$ is either S_i^1 or S_i^2. Similarly, the correct value for the ordered pair $(prev_j, curr_j)$, for process P_j, is either $R_j^1 = (curr_j - 1, curr_j)$ or $R_j^2 = (prev_j, prev_j + 1)$.

Let $(prev'_i, curr'_i) \in S_i^u$ and $(prev'_j, curr'_j) \in R_j^v$, where $u, v \in \{1, 2\}$. (S_i^u, R_j^v) is correct for some $u, v \in \{1, 2\}$ if and only if one of *Conditions* 4, 5 or 6 is *True*.

Condition 4 $((prev'_i = prev'_j + 1) \wedge (curr'_i = curr'_j + 1) \wedge \neg(state\_curr_i = state\_curr_j) \wedge (state\_curr_i = T))$

Condition 5 $((prev'_j = prev'_i + 1) \wedge (curr'_j = curr'_i + 1) \wedge \neg(state\_curr_i = state\_curr_j) \wedge (state\_curr_j = T))$

Condition 6 $((prev'_j = prev'_i) \wedge (curr'_j = curr'_i) \wedge (state\_curr_i = state\_curr_j))$

If P_j is undecided, it forwards the message to the next process, unchanged. If P_j is able to correct the fault it writes the corrected value in the appropriate variable, changes the message tag from U to D and then forwards the message to the next process.

When P_k receives a message with tag D, if it finds that $pred_1 = False$ then either $prev_k$ or $curr_k$ has a data fault. P_k can correct the fault according to Procedure *LocalCorrection(i,k)*.

```
Procedure LocalCorrection(i,k)
begin
    if ($(state\_curr_k = P) \wedge (state\_curr_i = P)$) then
        $curr_k \leftarrow curr_i$ and $prev_k \leftarrow prev_i$
    else if ($(state\_curr_k = P) \wedge (state\_curr_i = T)$) then
            $curr_k \leftarrow curr_i - 1$ and $prev_k \leftarrow prev_i - 1$
        else if ($(state\_curr_k = T) \wedge (state\_curr_i = P)$) then
            $curr_k \leftarrow curr_i + 1$ and $prev_k \leftarrow prev_i + 1$
        end if
    end if
end
```

After correcting the data fault, if $curr_k < curr_i$, P_k takes a temporary checkpoint with $v\_no = curr_i$ and then processes the message. This case arises if and only if the application message was sent after the sender has taken a new temporary checkpoint with a $v\_no = curr_i$. If $curr_k \geqslant curr_i$, P_k processes the message without taking a checkpoint. After processing the message P_k sends an acknowledgement message (*ack_msg*) with $state\_curr_i$, $curr_i$, $curr_k$ to P_i.

If P_j receives a message with tag D and finds that $pred_1$ is *False*, then it corrects the data fault using Procedure *LocalCorrection(i,j)* and forwards the message to the next process.

On receiving an *ack_msg* from P_k, process P_i first makes its correction if $pred_1 = False$. Then it deletes the corresponding message logged in the *MLT*. When P_k receives a message with tag U from P_i, if $pred_1 = False$ and one of *Conditions* 1, 2 or 3 is *True*, then P_k also becomes undecided. P_k keeps the message for future processing. It passes the message without message data to the next process with i as the changed receiver id of the message.

In the worst case, a message with tag U returns back to P_i, the originator of the message. If there exists at least one i such that $state\_curr_i = T$, P_i will wait for *ckpt_req*. After receiving *ckpt_req*, P_i corrects the data fault. Otherwise, all processes have data faults and they are unable to rectify these faults. In this case *global reset* is required. Several processes may receive such messages with tag U returned to them. Another round of message passing is required to elect one process among them (we choose the one with minimum id). This can be done by passing a message round the ring. In total, there will be $O(n)$ messages and will take $O(n)$ time. Let P_m be the elected process. As it is impossible to decide which one of $prev_m$ and $curr_m$ is correct, P_m assumes that $prev_m$ is correct. $curr_m$ is replaced by $prev_m + 1$. P_m sends a correction message (*correction_msg*) with $curr_m$ and $state\_curr_m$ to other processes.

On receiving *correction_msg*, P_j corrects itself with Procedure *LocalCorrection(m,j)*. The *correction_msg* is forwarded until it passes through all the processes and it returns back to P_i. The message which was held up due to U tag is processed after recovery.

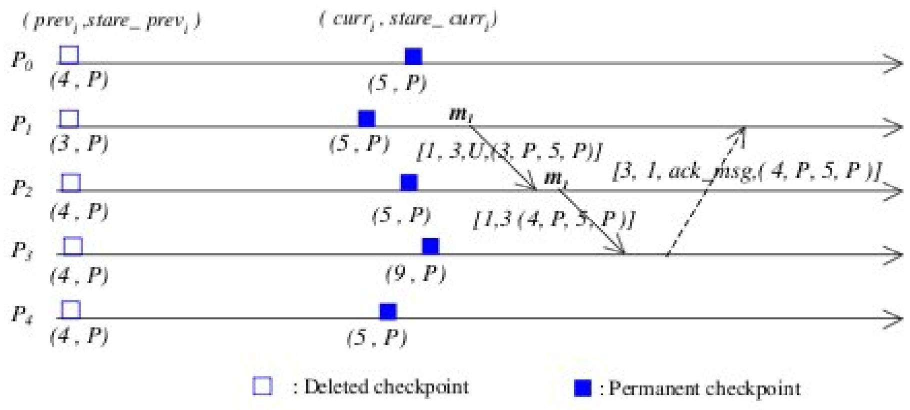


Fig. 2. An example showing data faults but not requiring global reset.

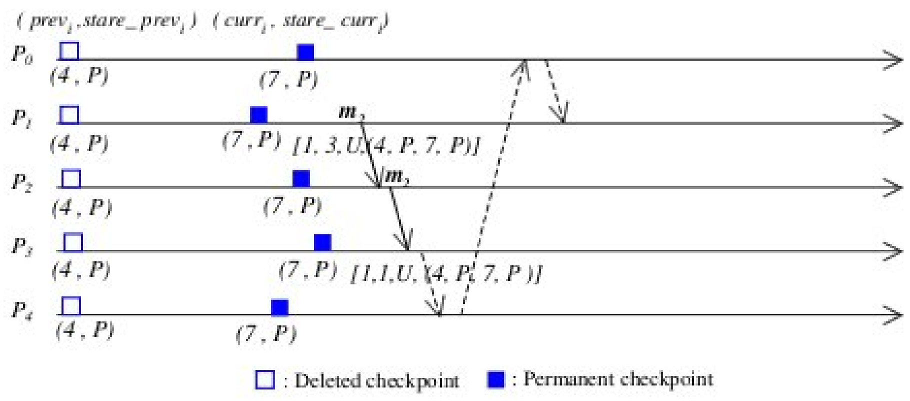


Fig. 3. An example showing data faults which requires global reset.

5.1. Examples

Our scheme is illustrated with the help of two examples. Fig. 2 shows a system consisting of five processes, P_0, P_1, P_2, P_3 and P_4. The ordered tuple ($prev_i$, $state\_prev_i$, $curr_i$, $state\_curr_i$) represents the four variables of P_i. P_1 checks its $pred_1$ and finds that $pred_1 = False$ before sending an application message m_1 to P_3. P_1 has a data fault. It cannot decide which one of (3, P, 4, P) or (4, P, 5, P) is correct. P_1 reaches an *undecided* state. P_1 sends m_1 with tag U and (3, P, 5, P) to P_2 mentioning that the destination is P_3. P_2 observes that m_1 has tag U. It checks its $pred_1$ and finds that $pred_1 = True$. P_2 corrects the fault using Procedure *LocalCorrection(2,1)*, and forwards m_1 to P_3 with tag D and corrected tuple (4, P, 5, P). On receiving m_1, P_3 checks its $pred_1$ and finds that $pred_1 = False$. P_3 corrects the fault using the same Procedure *LocalCorrection(1,3)*. Corrected tuple of P_3 would be (4, P, 5, P). P_3 than processes m_1 without taking any checkpoint since $curr_3 = curr_1$. After processing m_1 P_3 sends an *ack_msg* to P_1. On receiving the *ack_msg* P_1 rectifies its fault using Procedure *LocalCorrection(1,1)*.

Fig. 3 shows an example where global reset is necessary. This situation can happen only when all processes have same type of data faults, either $prev_i$ or $curr_i$ has been changed for all i with the same value. In Fig. 3 P_1 is trying to send a message, m_2 to P_3. Before sending m_2, P_1 finds that $pred_1 = False$ because $curr_1 \neq prev_1 + 1$. P_1 sends m_2 with tag U and (4, P, 7, P) to P_2 mentioning that the destination is P_3. When m_2 reaches P_2, P_2 finds that the message has tag U. P_2 checks its $pred_1$ and finds that $pred_1 = False$. P_2 tries to correct the faults using the order tuple (4, P, 7, P) carried by m_2 and its own order tuple (4, P,7, P). From those two order tuples P_2 finds two possible fault free order tuples, (4, P, 5, P) and (4, P, 5, P) or (6, P, 7, P) and (6, P, 7, P) for P_1 and its own, respectively, but cannot decide which would be the correct with respect to the global state. P_2 also reaches an *undecided* state, forwards m_2 to P_3. P_3 also cannot break the deadlock. It keeps the message part of m_2 in its log and forwards the header to P_4 mentioning that the destination is P_1. With the progress of the message, P_4 and P_0, reach *undecided* state. Finally, m_2 returns back to P_1. P_1 then sends one round of message for global resetter election. P_1 is elected as a global resetter since it is the only process in the election in this example. P_1 resets the tuple to (4, P, 5, P) from (4, P, 7, P) and sends *correction_msg* to all processes. P_i corrects the fault using Procedure *LocalCorrection(1,i)*. Corrected order tuples for all P_i would be (4, P, 5, P).

6. Data fault detection and correction algorithms

Algorithm 6.1. *Application message sender* (P_i).

```
{A process performs following operation when it sends an application message
with the order tuple ($prev_i$, $state\_prev_i$, $curr_i$, $state\_curr_i$)}
1. begin
2.    if($pred_1 = True \wedge pred_2 = True$) then
3.          attach tag $D$ with the message and send
4.    else if ($pred_2 = True$) then
5.          corrects the fault by putting $state\_prev_i = P$
6.          attach tag $D$ with the message and send
7.    else if ($pred_1 = True$) then
8.          set $msg\_send\_undecided_i \leftarrow T$
9.          attach tag $U$ with the message and send
10.   end if
11.   log the message in $MLT$
12. end
```

Algorithm 6.2. *Forward application message by P_j, $j \neq k$ (message is U tagged).*

```
1. begin
2. if ($pred_1 = True$) then                                  {$P_j$ has no data fault}
3.    correct the tuple using Procedure LocalCorrection(j,i), change the tag from $U$
4.    to $D$, forward the message to the next process with the corrected tuple.
5. else if ($pred_1 = False$) then                             {$P_j$ has a data fault}
6.    if ($Condition\ 1 = True \vee Condition\ 2 = True \vee Condition\ 3 = True$) then
7.          {$P_j$ cannot correct the fault}
8.          forward the message to the next process as it is.
9.    else $P_j$ corrects the fault using $Condition\ 4\ or\ 5\ or\ 6$
10.         changes the tag from $U$ to $D$, forwards the message to
11.         the next process with the corrected tuple.
12.   end if
13. end if
14. end
```

Algorithm 6.3. *Forward application message by P_j, $j \neq k$ (message is D tagged).*

```
1. begin
2.    if ($pred_1 = False$) then                               {$P_j$ has a data fault}
3.          $P_j$ corrects its own fault using Procedure LocalCorrection(i,j) .
4.    end if
5.    forward the message to the next process as it is.
6. end
```

Algorithm 6.4. *Receive application message by P_k (message is U tagged).*

```
1. begin
2. if ($pred_1 = True$) then                                   {$P_k$ has no data fault}
3.    correct the tuple using Procedure LocalCorrection(k,i),
4.    COMPARE($curr_k$, $curr_i$);
5. else                                                          {$P_k$ has a data fault}
6.    if ($Condition\ 1 = True \vee Condition\ 2 = True \vee Condition\ 3 = True$) then
7.                                                          {$P_k$ cannot correct the fault}
8.          keep the message in the log as withheld message and forward the header
9.          to the next process informing that the destination is $P_i$
10.   else $P_k$ correct the fault of its own and the tuple using $Condition\ 4\ or\ 5\ or\ 6$
11.         COMPARE($curr_k$, $curr_i$);
12.   end if
13. end if
14. end
```

Algorithm 6.5. *Receive application message by P_k (message is D tagged).*

```
1. begin
2.    if ($pred_1 = False$) then                              {$P_k$ has a data fault}
3.        $P_k$ corrects its own fault using Procedure LocalCorrection(i,k)
4.    end if
5.    COMPARE($curr_k, curr_i$);
6. end
```

Algorithm 6.6. *COMPARE($curr_k$, $curr_i$).*

```
1. begin                                       {may take checkpoint and sends
2.    if ($curr_k < curr_i$) then                          acknowledgement}
3.        take a checkpoint with $state\_curr_k = T$
4.        set $v\_no \leftarrow curr_i$ and $curr_k \leftarrow curr_i$
5.    end if
6.    process the message, send ack_msg with corrected tuple to $P_i$ with $curr_k$.
7. end
```

Algorithm 6.7. *Receive ack_msg / U / D Message by P_i.*

```
1. Acknowledgement (ack_msg):
1.1 begin
1.2   if ($pred_1 = False$) then                              {$P_i$ has a data fault}
1.3       correct its own fault using Procedure LocalCorrection(j,i)
1.4   if ($receive\_interval + 2 \leqslant curr_i$) then
1.5       delete the message from MLT
1.6   else $receive\_interval \leftarrow curr_i$      {received interval is maintained in MLT}
1.7   end if                                                      {for a message}
1.8 end

2. Message has tag D:          {$P_k$, message receiver could not correct the fault}
2.1 begin                      {some other process corrected the tuple of the message}
2.2   correct its own fault using Procedure LocalCorrection(j,i) and sends the
2.3   message with tag D towards destination $P_k$.
2.4 end

3. Message has tag U:          {all process have exactly the same fault as $P_i$}
3.1 begin                                   {none of the processes correct the fault}
3.2   ElectLeader.
3.3   if ($P_i$ is the leader) then
3.4       set $curr_i \leftarrow prev_i + 1$ and send correction_msg
3.5       to all processes with ($prev_i$, $state\_prev_i$, $curr_i$, $state\_curr_i$).
3.6   end if
3.7 end
```

Algorithm 6.8. *Receive correction_msg by P_j.*

```
1. begin
2.    correct the fault using Procedure LocalCorrection(i,j).
3.    if (there is any withheld message in log) then
4.        COMPARE($curr_j$, $curr_i$);
5.    end if
6.    Forward the correction_msg until it reaches $P_i$
7. end
```

7. Checkpointing algorithm

A process, P_i without a temporary checkpoint or any data fault may initiate checkpointing. During initiation P_i takes a temporary (T) checkpoint, sets a flag (*initiator_flag*$_i$) equal to *True* and increments v_no by one. All control messages for the checkpointing are routed in the *anti-clockwise* direction. The following checks are carried out during the initiation.

Algorithm 7.1. *Checkpointing initiated by P_i.*

```
1. begin
2.   if $((pred_1 = True) \wedge (pred_2 = True) \wedge (state\_curr_i = P))$ then
3.        take a checkpoint
4.        set $initiator\_flag_i \leftarrow True$, $state\_curr_i \leftarrow T$,
5.        $prev_i \leftarrow curr_i$, $curr_i \leftarrow curr_i + 1$, $v\_no \leftarrow curr_i$,
6.        send ckpt_req with $curr_i, i$
7.   end if
8. end
```

On receiving a *ckpt_req*, if P_j finds, both $pred_1$ and $pred_2$ are *True* then $curr_j$ is compared with the $curr_i$ of the message. If $curr_i$ is not equal to $curr_j$ and current checkpoint state, $state\_curr_j = P$ then P_j takes a new temporary checkpoint otherwise it do not take a new checkpoint. If one of the predicates, $pred_1$ and $pred_2$ is *False*, it implies that P_j has a data fault. It corrects the fault using the information tagged in the *ckpt_req* and takes a checkpoint according to the above condition as follows:

Algorithm 7.2. *On receiving ckpt_req by P_j.*

```
1. begin
2.   if $(pred_1 = True) \wedge (pred_2 = True)$ then
3.        if $(curr_j \neq curr_i) \wedge (state\_curr_j = P)$ then take a checkpoint
4.             set $state\_curr_j \leftarrow T$, $curr_j \leftarrow curr_i$, $prev_j \leftarrow curr_j - 1$
5.             $v\_no \leftarrow curr_j$, $initiator\_flag_j \leftarrow False$
6.        else do not take checkpoint
7.        end if
8.   else if $(pred_1 = True) \vee (pred_2 = True)$ then
9.        if $(pred_2 = False)$ then $state\_prev_j = P$
10.       else set $curr_j \leftarrow curr_i$, $prev_j \leftarrow curr_j - 1$
11.       end if
12.       if $(state\_curr_j = P)$ then take a checkpoint
13.            set $curr_j \leftarrow curr_i$, $prev_j \leftarrow curr_j - 1$, $v\_no \leftarrow curr_j$
14.            $initiator\_flag_j \leftarrow False$, $state\_curr_i \leftarrow T$
15.       end if
16.  end if
17. end
```

As concurrent multiple initiations of checkpointing are allowed, several *ckpt_req* generated by different initiators may be received by a process. Among them only initiators take decision to forward, discard or generate a commit message (*commit_msg*) to the next process along the anti-clockwise direction according to the following logic.

Algorithm 7.3. *ckpt_req propagation by P_j.*

```
1. begin
2.   if $(initiator\_flag_j = True) \wedge (j < initiator\_id)$ then discard the message
3.   else if $(initiator\_flag_j = True) \wedge (j = initiator\_id)$ then discard the message,
4.        send a commit_msg to the next process.
5.   else if $(initiator\_flag_j = True) \wedge (j > initiator\_id)$ then
6.        forward the ckpt_req to the next process.
7.   end if
8. end
```

On receiving a *commit_msg*, P_j takes the following actions:

Algorithm 7.4. *commit_msg propagation by P_j.*

```
1. begin
2.   if $(j \neq i)$ then
3.        delete the checkpoint with $v\_no = prev_j$, keeping $prev_j$ unchanged.
4.        set $state\_curr_j \leftarrow P$, forward the commit_msg to the next process.
5.   end if
6. end
```

When the *commit_msg* returns back to its creator, it stops the message propagation. The checkpointing process is terminated and a CGS, one checkpoint per process with same *v_no* is established. The checkpointing algorithm would not have been successful if at least one process was unable to take checkpoint for the checkpointing progress.

8. Process fault recovery

If a process, P_i fails due to crash, it loses the data stored in volatile storage. Hence it loses checkpointing variables, $prev_i$, $state\_prev_i$, $curr_i$, $state\_curr_i$ stored in volatile storage. The process reconstructs the process state from a checkpoint saved in the non-volatile stable storage. Since, the data in the non-volatile stable storage remain unaffected in case of process fault, the system can resume computation consistently from a CGS. A process may fail any time when the distributed computation is in progress. A process may have two checkpoints, one permanent and one temporary, or a single permanent checkpoint stored in the stable storage. The temporary checkpoint is taken on receiving a *ckpt_req* or induced by communication. Recovery algorithm finds out one checkpoint per process in non-volatile stable storage with same $curr_i$ $\forall i \in \{0, 1, 2, \ldots, n-1\}$. CGS finding and resuming computation is the task for recovery algorithm. Data fault, if any, in the other processes is corrected during the execution of the recovery algorithm. The faulty process starts recovery process by sending recovery message with v_no tagged with it. This value of v_no is retrieved from the checkpoint stored in non-volatile stable storage.

Algorithm 8.1. *Recovery initiated by P_i.*

```
1. begin
2.    retrieve latest checkpoint from stable storage
3.    set $state\_prev_i \leftarrow P$, $state\_curr_i \leftarrow P$
4.        $curr_i \leftarrow v\_no, prev_i \leftarrow curr_i - 1, recovery\_id \leftarrow i$
5.    if only one checkpoint is in the stable storage then $recovery\_flag_i \leftarrow True$
6.    else $recovery\_flag_i \leftarrow False$    {two checkpoints are stored in stable storage}
7.                        {the latest checkpoint may or may not belong to the $CGS$}
8.    end if
9.    send recovery message with $curr_i$, $recovery\_flag_i$ and $recovery\_id$
10. end
```

Algorithm 8.2. *Recovery message received by P_j.*

```
1. begin
2.    if $j \neq i$ then
3.        if $recovery\_flag_i = True$ then
4.            if $pred_1 = False \vee pred_2 = False$ then
5.                set $state\_prev_i \leftarrow P$, $state\_curr_i \leftarrow P$
6.                    $curr_j \leftarrow curr_i, prev_j \leftarrow curr_j - 1$
7.            if two checkpoints are stored in stable storage then
8.                delete the latest checkpoint
9.            resume computation from the checkpoint stored in stable storage
10.       else                                          {$recovery\_flag_i = False$}
11.           if $pred_1 = False$ then
12.               if $state\_curr_j = T$ then
13.                   $curr_j \leftarrow curr_i, prev_j \leftarrow curr_j - 1$
14.                   forward the recovery message
15.               else                                  {$state\_curr_j = P$}
16.                   $recovery\_id \leftarrow j$            {reset recovery initiator}
17.                   $curr_j \leftarrow curr_i - 1, prev_j \leftarrow curr_j - 1$
18.                   $recovery\_flag_j \leftarrow True$, send recovery message
19.               end if
20.           else                                      {$pred_1 = True$}
21.               if $curr_j = curr_i$ then
22.                   forward the recovery message
23.               else if $curr_j > curr_i$ then
24.                   delete the latest checkpoint
25.                   $curr_j \leftarrow curr_i, prev_j \leftarrow curr_j - 1$
26.               else                                  {$curr_j < curr_i$}
27.                   $recovery\_id \leftarrow j$ {reset recovery initiator}
28.                   $recovery\_flag_j = True$, send recovery message
29.               end if
30.           end if
31.       end if
32.   else {j = i}
33.       $P_j$ resumes computation from the latest checkpoint
34.       stored in stable storage
35.   end if
36. end
```

9. Correctness and complexity analysis

A set of checkpoints is consistent if it does not contain any orphan or missing message. In this section we first show that a set of checkpoints with the same version number will have no orphan message. Though the system may contain missing messages, but every such message is available in the *MLT* of the sender.

Lemma 1. *There will not be any orphan message in the system.*

Proof. Suppose there is an orphan message (m) in the system. A message is an orphan message if it is send by its sender P_i after taking checkpoint α (say) but is received and processed by its receiver P_k, while it is yet to take checkpoint α. We need to consider whether P_i, at the time of sending the message, and P_k, at the time of receiving the message, has any data fault. The following cases may arise:

Case 1: Suppose neither P_i nor P_k has a data fault. Last checkpoint version number ($curr_i = \alpha$) of P_i is tagged with m by Algorithm 6.1. After receiving m, P_k finds that the last checkpoint version number is less than α and the checkpoint version number tagged with m is α, i.e., $curr_k < curr_i$ (Algorithm 6.6, step 2). So P_k takes a temporary checkpoint with $v\_no = \alpha$ and then processes m by Algorithm 6.6, steps 2–6. Which contradicts our assumption that the v_no of P_k is less than α at the time of processing m.

Case 2: Suppose P_i has a data fault and but P_k has no data fault. P_i sends an undecided (U) tag with m (Algorithm 6.1, step 9). If m arrives at P_k with U tag (Algorithm 6.4) it corrects the fault (step 3) and takes decision for new checkpoint similar to Case 1. So, again the v_no of P_k is α at the time of processing m.

Case 3: Suppose P_i has no data fault but P_k has a data fault. In this case, P_i sends a D tag with m (Algorithm 6.1, step 3) to P_k. P_k corrects the fault (Algorithm 6.5, step 3) then takes decision for new checkpoint according to Case 1.

Case 4: Suppose there are data faults in both P_i and P_k. P_i sends a U tag with m (Algorithm 6.1, step 9). If m arrives at P_k with D tag, the arguments of Case 3 hold good. If m arrives with tag U, P_k is unable to correct the fault (Algorithm 6.4, step 6). In this case P_k keeps the message in the log as withheld message and forwards the header to next process informing that the destination is P_i. If the header with D tag returns back to P_i (Algorithm 6.7, step 2) it corrects faults and sends the header with D tag towards destination P_k. If the header with U tag returns back to P_i (Algorithm 6.7, step 3) a leader is elected. The leader corrects fault and sends correction message to all other process (Algorithm 6.7, step 3.4 and 3.5) including P_k. Then P_k corrects the fault and takes decision for new checkpoint and processing the withheld message (Algorithm 6.8) according to Case 1. So once again, at the time of processing m, P_k has $v\_no = \alpha$. Contradiction! □

Lemma 2. *All missing messages are available in the MLTs of the senders.*

Proof. Suppose there is a missing message (m') in the system. A message is a missing message if it is send by its sender P_i before taking checkpoint β (say) but is received and processed by its receiver, P_k after taking checkpoint β. If neither P_i nor P_k has a data fault, on receiving m', P_k sends *ack_msg* to P_i with $curr_k$ (current checkpoint interval). On receiving the *ack_msg*, P_i deletes the message from *MLT* (Algorithm 6.7 step 1.5) only when a permanent checkpoint has been taken by the receiver after processing the message.

Note that by Algorithm 6.7 in steps 1.4 and 1.6, the variables used are guaranteed to be error-free, even if P_i and/or P_k has data faults. If there was a data fault in P_i, it is corrected in step 1.3 and if there was a data fault in P_k, it is corrected by Algorithm 6.5 in step 3. If there are data faults in both P_i and P_k, faults are corrected as per the arguments given in the proof for Lemma 1 in Case 4. So, like the fault-free case, a message is deleted only when the sender is sure that the receiver has taken a permanent checkpoint after processing the message. □

Lemma 3. *In case of data faults, the system corrects the data faults in $O(n)$ steps* (*one step is one hop message communication*).

Proof. Following the arguments given in the proof for Lemma 1, we can prove that data faults would be corrected. In Case 1, neither P_i nor P_k have data faults; so it need not be considered. In Case 2, P_i has a data fault and but P_k has no data fault. After receiving the message, P_k sends *ack_msg* with corrected tuple to P_i. On receiving the *ack_msg*, P_i corrects the fault (Algorithm 6.7, step 1.3). The correction is completed in $O(n)$ steps. In Case 3, P_i has no data fault but P_k has a data fault. P_k receives the message with tag D and corrects the fault (Algorithm 6.5, step 3). The correction is completed in $O(n)$ steps. In Case 4, if the header returns to P_i with tag D (Algorithm 6.7, step 2), then the correction is completed in $2n$ steps. If it returns with tag U (Algorithm 6.7, step 3), leader election takes an extra n steps. So totally $3n$ steps are required. □

Theorem 1. *The set of checkpoints generated by the proposed checkpointing algorithm is consistent. The time and message complexities of this algorithm are both $O(n)$.*

Proof. Lemmas 1 and 2 guarantee that the set of checkpoints is consistent.

Let P_i be the process with the minimum id among the concurrent initiators. P_i will eventually send its *ckpt_req* in anti-clockwise direction, and since no other initiator or non-initiator process can discard the message, the message will return back to P_i. Then P_i sends a *commit_msg*. The *ckpt_req* of every other initiator will be discarded when next initiator of smaller id receives the message in anti-clockwise direction of the ring. Each initiator process will forward at most one message. At most $n + (n - 1)$ *ckpt_req* messages may be generated for concurrent initiations. Another n messages would be required for the *commit_msg* to return back to P_i. So the total number of messages is at most $3n - 1$.

It is assumed that a message takes one unit of time to travel across one link. We ignore the time for processing a message at a process. After n time units the *ckpt_req* returns back to P_i. Another n units of time are required for the *commit_msg* to return back to P_i. In total after $2n$ units of time, the *CGS* is confirmed. □

Theorem 2. *Recovery algorithm ensures that the system returns to a legitimate state in $O(n)$ steps.*

Proof. In case of a process failure, the faulty process, on being revived, retrieves the latest checkpoint version number from the stable storage (Algorithm 8.1) and sends recovery message to all other processes for finding checkpoints with the same version number. If the checkpoint exists for all processes (Algorithm 8.2, step 32) then the CGS is established. Otherwise, previous checkpoint version number exists for all other processes and the CGS is established by Algorithm 8.2, step 15 in case of the receiver have data fault or Algorithm 8.2, step 26 in case of process do not have data fault. □

10. Probabilistic analysis of the algorithms

In this section, we study the performance of the proposed scheme under a stochastic model. Every process has four variables $prev_i$, $curr_i$, $state\_prev_i$, and $state\_curr_i$. A *data fault* in a process signifies that one of the variable is changed because of the unreliable system.

Let p be the probability of a fault (data fault) for a variable. We assume that faults in different variables are independent of each other.

10.1. Robustness of the system

There are a total of $4n$ variables in the n processes. The probability that all these processes are fault free (and hence the system is fault free) is

$$= (1 - p)^{4n}.$$

The above term represents the probability of the system being functional, for checkpointing scheme without self-stabilization.

In the proposed scheme, the system is functional if every process has at most one fault. For an arbitrary process, the probability that it has at most one fault (i.e., it has 0 or 1 fault) is

$$\begin{aligned} &= (1 - p)^4 + 4p\,(1 - p)^3 \\ &= (1 - p)^3(1 + 3p). \end{aligned}$$

The probability that all the n processes have at most one fault each is

$$= \left((1 - p)^3(1 + 3p)\right)^n.$$

Incremental robustness (IR) of the system is

$$= \left((1 - p)^3(1 + 3p)\right)^n - (1 - p)^{4n}.$$

Fig. 4 shows the value of the IR for various values of n and p. The probability of data fault in a variable is expected to be low. In such a case the gain in the robustness of the system can be pretty high.

10.2. Probability for global reset

Global reset is required when all processes have data faults of the same type and they are unable to rectify these faults. In such a case, for some s and t, $(s \neq t + 1)$, $prev_i = t$ and $curr_i = s\ \forall : i$ such that $0 \leqslant i \leqslant n - 1$. This implies that either $prev_i$ or $curr_i$ $\forall i \in \{0, n - 1\}$ has been changed. In this case, a message with tag U returns back to its originator.

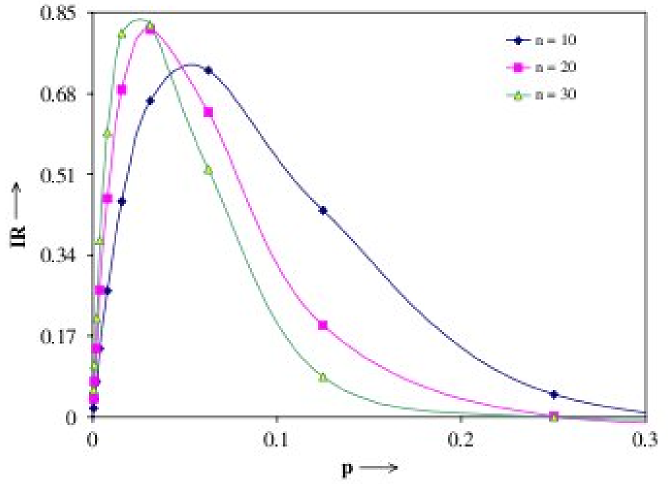


Fig. 4. Graph showing the incremental robustness of a system for different number of processes and under different values of p.

Table 1
Probability of global reset (PGR) for different values of p, n, and k

$\frac{1}{p} \rightarrow$	2	8	16	32	64
for $k = 30$					
$n = 10$	1.25×10^{-25}	2.34×10^{-24}	1.81×10^{-26}	4.72×10^{-29}	7.46×10^{-32}
$n = 20$	2.71×10^{-52}	9.42×10^{-50}	5.64×10^{-54}	3.85×10^{-59}	9.58×10^{-65}
$n = 30$	5.86×10^{-79}	3.80×10^{-75}	1.76×10^{-81}	3.13×10^{-89}	1.23×10^{-97}
for $n = 10$					
$k = 40$	8.71×10^{-27}	1.62×10^{-25}	1.26×10^{-27}	3.28×10^{-30}	5.18×10^{-33}
$k = 20$	1.12×10^{-27}	2.08×10^{-26}	1.61×10^{-28}	4.21×10^{-31}	6.64×10^{-34}
$k = 30$	2.10×10^{-28}	3.91×10^{-27}	3.03×10^{-29}	7.91×10^{-32}	1.25×10^{-34}

Probability that P_0 has data fault in exactly one of the two variables, $curr_0$ and $prev_0$ among the four variables is

$$= 2\,p\,(1-p)^3.$$

For any process P_i $(1 \leqslant i \leqslant n-1)$ the probability that P_i has exactly the same fault as P_0 is

$$= (1-p)^3 \left(\frac{p}{k-1}\right),$$

where k is the maximum v_no for a checkpoint.

Probability of global reset (PGR) is

$$= 2(1-p)^{3n} \left(\frac{p^n}{(k-1)^{n-1}}\right).$$

The proposed scheme blocks process only when an election algorithm is run; which is the case when global reset is required. Table 1 shows that for all practical purposes, the proposed scheme may be considered non-blocking.

11. Comparison with existing algorithms

Concurrent checkpointing algorithms proposed by Spezialetti and Kearns [18], Prakash and Singhal [15] are designed for general network topologies. Their worst case message complexities are $O(n^3)$. But for the ring topology, this worst case is achieved. The algorithm proposed by Mandal and Mukhopadhyaya [12] can handle concurrent initiations of snapshot collection for unidirectional and bidirectional rings. The worst case message and time complexities are $O(n^2)$ and $O(n)$, respectively. The message and time complexities of this proposed algorithm are both $O(n)$.

Table 2
Performance of the proposed algorithm and other existing algorithms

	[18]	[15]	[12]	Proposed algorithm
Network topology for which applicable	General	General	Ring	Ring
Worst case message complexity in ring	$O(n^3)$	$O(n^3)$	$O(n^2)$	$O(n)$
Time complexity in ring	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Control message size for k concurrent initiations	$O(n/k)$	$O(n)$	$O(1)$	$O(1)$
Number of checkpoints (each process) stores for k concurrent initiations	One permanent one temporary	One permanent k temporary	One permanent one temporary	One permanent one temporary

Table 2 compares of the proposed algorithms with Spezialetti and Kearns [18], Prakash and Singhal [15], Mandal and Mukhopadhyaya [12] algorithms.

12. Conclusion

In this paper, self-stabilizing data fault detecting and correcting checkpointing and recovery algorithms have been proposed for an unreliable distributed system on a ring topology. We have considered data faults in the variables used by the checkpointing algorithm. The proposed algorithms can deal with data faults in multiple processes but at most one data fault per process. The recovery algorithm can reconstruct a failed process, maintaining consistency in the system, even in the presence of data faults in other processes. All the above algorithms converge in $O(n)$ steps. When all processes have data faults, but no process is able to correct the fault, a *global reset* is required. Global reset requires election of a leader and hence $O(n)$ extra overhead. It has been shown that PGR is too small. As the probability of data fault in a variable is expected to be low, we have shown that in such a case, the gain in the *robustness* of the system can be pretty high.

The proposed data fault detection and correction algorithms can be extended to work for a general network topology. The control message needs to traverse a route back to the originator and covering all the processes. Similarly, an application message with tag U will traverse a normal path to the destination. If it still has tag U, it will take a path to the originator covering the rest of the nodes. If its tag is changed to D, it can take a shortest path to originator, without covering the rest. An interesting extension will be to consider multiple data faults per process.

References

[1] L. Alvisi, B. Hoppe, K. Marzullo, Causality tracking in causal message-logging protocols, Distrib. Comput. 15 (2002) 1–15.
[2] G. Cao, M. Singhal, On coordinated checkpointing in distributed systems, IEEE Trans. Parallel Distribu. Syst. 9 (12) (1998) 1213–1225.
[3] G. Cao, M. Singhal, Checkpointing with mutable checkpoints, Theoret. Comput. Sci. 290 (2) (2003) 1127–1148.
[4] K.M. Chandy, L. Lamport, Distributed snapshots: determining global states of distributed systems, ACM Trans. Comput. Syst. 3 (1) (1985) 63–75.
[5] E.W. Dijkstra, Self stabilizing systems in spite of distributed control, Commun. ACM 17 (1974) 643–644.
[6] E.N. Elnozahy, L. Alvisi, Y.-M. Wang, D.B. Johnson, A survey of rollback-recovery protocols in message-passing systems, ACM Comput. Surveys 34 (3) (2002) 375–408.
[7] S. Ghosh, A. Gupta, T. Herman, S.V. Pemmaraju, Fault-containing self-stabilizing algorithms, in: Proceedings of the 15th ACM Symposium on Principles of Distributed Computation, 1996, pp. 45–54.
[8] S. Ghosh, X. He, Scalable self-stabilization, J. Parallel Distrib. Comput. 62 (5) (2002) 945–960.
[9] T. Herman, Self-stabilization bibliography: access guide, Chicago J. Theoret. Comput. Sci. Working Paper WP-1, URL: ⟨http://www.cs.uiowa.edu/ftp/selfstab/bibliography/⟩. 2002.
[10] D. Johnson, W. Zwaenepoel, Recovery in distributed systems using optimistic message logging and checkpointing, J. Algorithms 3 (11) (1990) 462–491.
[11] R. Koo, S. Toueg, Checkpointing and rollback-recovery for distributed system, IEEE Trans. Software Engrg 13 (1) (1987) 23–31.
[12] P.S. Mandal, K. Mukhopadhyaya, Concurrent checkpoint initiation and recovery algorithms on asynchronous ring networks, J. Parallel Distrib. Comput. 64 (5) (2004) 649–661.
[13] D. Manivannan, M. Singhal, Quasi-synchronous checkpointing: models, characterization, and classification, IEEE Trans. Parallel Distrib. Syst. 10 (7) (1999) 703–713.
[14] D. Manivannan, M. Singhal, Asynchronous recovery without using vector timestamps, J. Parallel Distrib. Comput. 62 (12) (2002) 1695–1728.
[15] R. Prakash, M. Singhal, Maximal global snapshot with concurrent initiators, in: Proceedings of the Sixth IEEE Symposium on Parallel and Distributed Processing, October 1994, pp. 334–351.
[16] M. Schneider, Self-stabilization, ACM Comput. Surveys 25 (1) (1993) 45–67.

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

[18] M. Spezialetti, P. Kearns, Efficient distributed snapshots, in: Proceedings of the Sixth International Conference on Distributed Computing Systems, 1986, pp. 382–388.

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

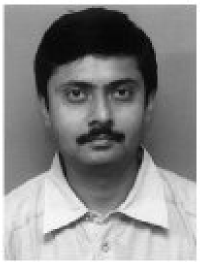
[20] N.H. Vaidya, Consistent logical checkpointing, Technical Report 94-051, Department of Computer Science, Texas A&M University, July, 1994.

[21] N.H. Vaidya, Staggered consistent checkpointing, IEEE Trans. Parallel Distrib. Syst. 10 (7) (1999) 694–702.

[22] Y.M. Wang, Y. Huang, W.K. Fuchs, Progressive retry for software error recovery in distributed systems, in: 23rd Annual International Symposium on Fault-Tolerant Computing, June 1993, pp. 138–144.

[23] Y.M. Wang, A. Lowry, W.K. Fuchs, Consistent global checkpoints based on direct dependency tracking, Inform. Process. Lett. 50 (4) (1994) 223–230.

Partha Sarathi Mandal received his Bachelor of Science (Hons.) in Mathematics from the University of Calcutta, India, in 1995. He received his Master of Science degree in Mathematics and Ph.D. in Computer Science from Jadavpur University, India, in 1997 and 2006, respectively. He is awarded Junior and Senior Research Fellowship by the Council of Scientific and Industrial Research (CSIR), India. He was a research fellow in Computer Science at the Advanced Computing and Microelectronics Unit of the Indian Statistical Institute. He was a recipient of the Young Scientist Award of the Indian Science Congress Association in the section of Information and Communication Science and Technology (including Computer Sciences) and Postdoctoral Fellowship of the Institut National de Recherche en Informatique et en Automatique (INRIA), France. He is currently doing his postdoctoral research with the Grand-Large team of INRIA Futurs at the Laboratoire de Recherche en Informatique (LRI) of the University Paris Sud, Orsay, France. His current research interests include wireless sensor networks, distributed computing, fault tolerance, mobile agents, performance analysis, self-stabilization, etc.



Krishnendu Mukhopadhyaya received his 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. He has served as a member of the technical program committees of international conferences like HiPC, VTC, etc.