Journal of Computing and Information Technology - CIT 16, 2008, 1, 23–33
doi:10.2498/cit.1000906

Design of High Performance Distributed Snapshot / Recovery Algorithms for Ring Networks

Bidyut Gupta[1], Shahram Rahimi[1] and Ziping Liu[2]

[1]Department of Computer Science, Southern Illinois University, Carbondale, Illinois, USA
[2]Department of Computer Science, Southeast Missouri State University, Cape Girardeau, Missouri, USA

In this work, we have presented non-blocking checkpointing and recovery algorithms for bidirectional networks. We have deviated from the conventional approach of taking first temporary checkpoints and then converting them to permanent ones by processes (as followed by any coordinated checkpointing scheme). Thus, the proposed coordinated checkpointing algorithm allows processes to take permanent checkpoints directly, without taking temporary checkpoints and whenever a process is busy, it takes a checkpoint after completing its current procedure. We have shown that the presented algorithms take much less time for their execution and use much less number of control messages (and hence much less number of interrupts to a process) when compared to a noted recent work [4].

Keywords: ring networks, coordinated checkpointing, recovery, distributed systems

1. Introduction

Checkpointing / rollback-recovery strategy has been an attractive approach for providing fault-tolerance to distributed applications [1] – [9]. A checkpoint is a snapshot of the local state of a process, saved on local nonvolatile storage to survive process failures. A global checkpoint of an n-process distributed system consists of n checkpoints (local) such that each of these n checkpoints corresponds uniquely to one of the n processes. A global checkpoint M is defined as a consistent global checkpoint / state (CGS) if no message is sent after a checkpoint of M and received before another checkpoint of M [1]. The checkpoints belonging to a consistent global checkpoint are called globally consistent checkpoints (GCCs).

Checkpointing algorithms may be classified into two main categories: (a) coordinated and (b) uncoordinated. In uncoordinated check pointing, each process takes checkpoint independently, without the knowledge of the other processes. In case of a failure after recovery, a CGS is found from the existing checkpoints and the system restarts from there.

In coordinated checkpointing, all processes synchronize through control messages before taking checkpoints. These synchronization messages contribute to extra overhead, but make the system free from domino effect. There are two types of coordinated check pointing algorithms: (a) blocking and (b) non-blocking. 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.

Most of the research in the area of coordinated checkpointing mainly concentrate on using the idea of non-blocking [3],[5],[7] – [10]. While most of the existing works put no restriction on the topologies of the distributed systems, there are special topologies for interconnecting the processing nodes in distributed systems, such as ring networks [4], [11]. The authors in [4] have proposed efficient checkpointing and recovery algorithms for ring networks. They have proposed a coordinated checkpointing algorithm in which processes take checkpoints independent of message pattern. It allows any process 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. Hence the algorithm runs faster than the algorithm proposed in [10]. The problem with this algorithm is that the number of control messages used is large and there is an overhead of taking a temporary checkpoint and then converting it into a permanent checkpoint.

The present work is aimed at designing high performance checkpointing and recovery algorithms for ring networks which outperform the similar algorithms reported recently in [4]. In [4], the authors have considered both unidirectional and bidirectional ring networks. However, their proposed checkpointing and recovery algorithms for unidirectional ring networks are trivial and a modified approach toward the same is to appear in [12]. Therefore, in this work we focus our attention on the bidirectional ring networks only. Since our objective is to design high performance checkpointing and recovery approaches compared to those in [4], we start with a brief and clear description of their checkpointing and recovery approaches. It may help in understanding clearly our 'problem formulation' which is stated later in this section.

In [4], the proposed algorithm for bidirectional ring networks works in the following way. Consider a ring network consisting of five processes P_0, P_1, P_2, P_3, and P_4 as shown in Figure 1a. Assume that the network is a bi-directional one and each process can send messages only to its predecessor and successor. For example, in this diagram process P_0 can communicate only to processes P_1 and P_4, process P_1 can communicate only to processes P_0 and P_2, and so on.

Without any loss of generality, let us assume that process P_2 initiates the checkpointing algorithm. Process P_2 first takes a temporary checkpoint $T_{2,1}$, and then sends checkpoint requests to its predecessor and successor processes P_1 and P_3 respectively. Each process, on receiving the first checkpoint request, takes a temporary checkpoint $T_{1,1}$ and $T_{3,1}$ respectively and forwards the request to the process from which it did not receive the request. Hence P_1 forwards the request to P_0, and P_3 forwards to P_4. P_0 and P_4on receiving the checkpoint request first take temporary checkpoints $T_{0,1}$ and $T_{4,1}$ and forward the message to P_4 and P_0 respectively.

P_4 receives the checkpoint request from P_0 and P_0 receives the checkpoint request form P_4. This checkpoint request message is the second request to both P_0 and P_4, hence both P_0 and P_4 convert their respective temporary checkpoints $T_{0,1}$ and $T_{4,1}$ to permanent checkpoints $C_{0,1}$ and $C_{4,1}$ respectively. After converting to permanent checkpoints, P_0 and P_4 forward the checkpoint request to P_1 and P_3 respectively. P_1 and P_3 on receiving the second checkpoint request convert their respective temporary checkpoints to permanent checkpoints $C_{1,1}$ and $C_{3,1}$ respectively and forward the checkpoint request to initiator process P_2. Suppose that process P_2 receives the checkpoint request from P_1 first. This message is the first checkpoint request to initiator P_2. Hence it converts its latest temporary checkpoint $T_{2,1}$ to permanent checkpoint $C_{2,1}$. Initiator process does not forward the checkpoint request any further. Eventually, it also receives the checkpoint request sent from process P_3, which it discards and the checkpointing algorithm is terminated. In Figure 1a it is shown that at time t the temporary checkpoint $T_{0,1}$ is converted into a permanent checkpoint, denoted here as $C_{0,1}$. The same is followed for all processes in the system.

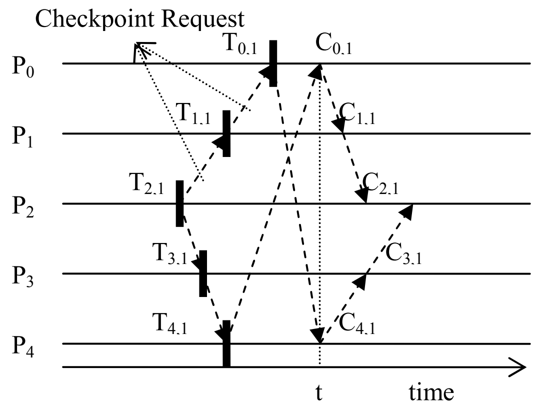

Figure 1a. An example of the checkpointing algorithm.

In Figure 1b, we describe the working principle of the recovery algorithm. Without any loss of generality, let us suppose that process P_4 fails during the execution of the checkpointing algorithm. Suppose that, by the time failure occurs, all processes have taken temporary checkpoints $T_{0,1}$, $T_{1,2}$, $T_{2,1}$, $T_{3,1}$, and $T_{4,1}$. After recovering from the failure P_4 starts the recovery algorithm.

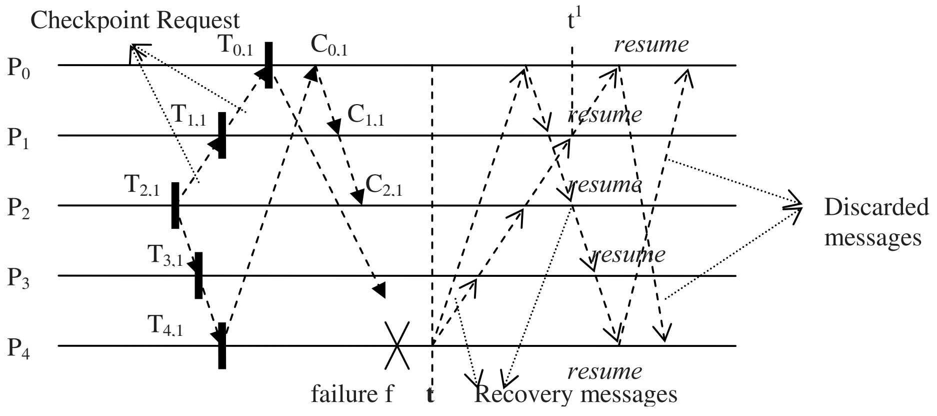

Figure 1b. An example of the recovery algorithm.

Assume that before P_4 starts the recovery algorithm, processes P_0, P_1, and P_2 have converted their respective temporary checkpoints to permanent checkpoints $C_{0,1}$, $C_{1,1}$ and $C_{2,1}$. Process P_3 did not receive yet any message from P_4 and hence it did not convert its temporary checkpoint to a permanent one.

After process P_4 recovers from its failure (say, at time t as shown in Figure 1b), it initiates the recovery algorithm. It sets its *flag_visit* flag to true and sends recovery message to its adjacent processes P_3 and P_0 with its latest checkpoint version number. On receiving a recovery message, each process sets its *flag_visit* flag to true and forwards the recovery message to the process from which it did not receive recovery message. Process P_3 forwards the message to P_2 and process P_0 to P_1. Assume that process P_2 receives the recovery message sent by P_3 before the recovery message from process P_1. Process P_2 sets its flag to true and forwards the message to process P_1. Process P_1, which receives the recovery message from P_0 first, sets its *flag_visit* flag and forwards the request to process P_2.

Later process P_1 receives the recovery message forwarded by process P_2, say at time t^1. This is the second such recovery message to process P_1. It observes that its *flag_visit* flag is already set to true. Therefore it first sets its *flag_resume* flag to true and then forwards the message to its predecessor process $P_0$and starts computation from its latest permanent checkpoint $C_{1,1}$. Eventually, process P_2 receives the recovery message sent by process P_1. This message is the second recovery message to P_2. It observes that its *flag_visit* flag is true. Therefore, it sets its *flag_resume* to true, forwards the message to P_3 and restarts the computation from its latest permanent checkpoint $C_{2,1}$.

Process P_0 receives the recovery message from P_1, it observes that its *flag_visit* flag is true. Therefore, it sets its *flag_resume* flag to true, and then forwards the message to its predecessor process P_4 and restarts computation from $C_{0,1}$.

Process P_3 receives the recovery message sent from P_2. This message is the second recovery message to process P_3. On receiving the recovery message, P_3 observes that its *flag_visit* flag is set to true. Hence it sets its *flag_resume* to true and then forwards the message to process P_4 and rolls back to its latest checkpoint. Since its latest checkpoint is a temporary checkpoint $T_{3,1}$, it first converts the temporary checkpoint $T_{3,1}$ to a permanent checkpoint $C_{3,1}$and restarts computation from that checkpoint.

Recovery messages are sent to process P_4 from P_0 and P_3. Suppose that P_4 receives the recovery message sent by process P_3 before it receives the recovery message from process P_0. This is the first recovery message to the initiator. Its *flag_visit* flag is already set. Hence, on receiving the recovery message from process P_3, it sets its *flag_resume* flag to true and forwards the recovery message to its successor process P_0. It observes that its latest checkpoint $T_{4,1}$is a temporary checkpoint. Therefore, it first converts its latest temporary checkpoint to a permanent checkpoint and then rolls back and restarts normal computation from that checkpoint. Later, process P_4 receives also the recovery message sent by process P_0. On receiving the recovery message, it checks its *flag_resume* flag. Since it is true, it means that process P_4 has already restarted its computation. Hence, this second recovery message is discarded. In the meantime, process P_0 receives the recovery message sent by process P_4. In a similar way it observes that its *flag_resume* flag is already set to true. Hence, it discards this recovery message.

In the above example, we note that the recovery message is forwarded by any process if and only if the *flag_resume* flag of the process is false. We also observe that during recovery, if the latest checkpoint of a process is found to be a temporary checkpoint, it is converted into a permanent checkpoint and then the computation is started from that checkpoint. In this context, it is worth mentioning that restarting during a process is delayed until the *flag_resume* flag becomes true. Besides, the numbers of control messages required for the checkpointing and the recovery algorithms are significantly large.

Problem Formulation

In the checkpointing algorithms proposed in [4], [5], [7], and [9], processes are supposed to take temporary checkpoints first, and then these checkpoints are converted to permanent ones and only then, these permanent checkpoints are considered to form a CGS of the system. However, during the execution of the checkpointing algorithm, if a process is busy with some high priority procedure, when a checkpointing request arrives at it, the process will not take a checkpoint. In such a situation, every process that has already taken a temporary checkpoint must discard it and continue normal execution. Later, the checkpointing algorithm has to be restarted again. The same problem may arise when a process receives a request to convert its temporary checkpoint to a permanent one while it is busy executing a high priority procedure. In this case also, the checkpointing algorithm has to restart later. Thus, such situations definitely affect the execution time of the application programs adversely.

To avoid this problem we have proposed a checkpointing algorithm which takes permanent checkpoints directly, without taking temporary checkpoints, and whenever a process is busy, the process takes a checkpoint after completing the current procedure. The proposed checkpointing and recovery algorithms are both single phase and non-blocking ones. We do not consider concurrent initiations of the algorithms, as it may cause message storm in large distributed systems. We have shown that the proposed algorithms take much less time for their execution and use much less number of control messages (and hence much less number of interrupts to a process) when compared to the algorithms in [4].

This paper is organized as follows. In Section 2 we have stated the system model, the relevant data structures, and a formal description of the proposed checkpointing algorithm along with an illustration. In Section 3 we have stated the recovery algorithm along with an illustration. Also, in this section we have discussed the performance of the proposed algorithms. In Section 4 we have stated how the proposed basic checkpointing algorithm can be modified so that only minimum number of processes will take checkpoints, which has not been addressed in [4].

2. System Model, Data Structures, and the Checkpointing Algorithm

2.1. System Model

We consider that the bidirectional ring network architecture consists of n processes P_0, P_1, P_2, ..., P_{n-1}. These processes do not share memory and they communicate via messages. We also assume that the i^{th} process can directly send a message to j^{th} process if and only if $j = (i+1) \bmod n$ or $j = (i-1) \bmod n$. That is j^{th} process is the immediate successor to i^{th} process if and only if $j = (i+1) \bmod n$ and j^{th} process is the immediate predecessor to i^{th} process if and only if $j = (i-1) \bmod n$. The communication channel is assumed to be first in first out (FIFO).

2.2. Data Structures

Consider a distributed system of n processes P_0, P_1, P_2, ..., P_{n-1} involved in the execution of a distributed algorithm on a bidirectional ring network. In a bidirectional ring network a process P_i can send an application or control message only to its adjacent process P_j where $j = (i+1) \bmod n$ or $(i-1) \bmod n$.

A checkpoint sequence number denotes the number of times the checkpointing algorithm has been executed so far. In other words, if the checkpoint sequence number is k, it means that the checkpoint is taken on the k^{th} execution of the checkpointing algorithm. Initially, checkpoint sequence number for each process is set to zero.

In our algorithms, we use two kinds of control messages. The first one is a checkpoint request $< C_p, i >$, which as in [4], is sent by an initiator process P_i (to initiate the checkpointing algorithm) to its adjacent processes, where i is the process id of the initiator process. The second control message is the recovery message $< R_c, j >$ used in the recovery approach. The recovery algorithm is initiated by the process which has recovered from a failure. This process initiates the recovery algorithm by sending this recovery message to its adjacent processes, asking them to restart from their respective checkpoints taken during the j^{th} execution of the checkpointing algorithm.

2.3. An Illustration

We now explain with an example how our proposed checkpointing approach works. Consider a distributed system of five processes P_0, P_1, P_2, P_3, and P_4 in a bidirectional ring network as shown in Figure 2. Assume that process P_2 initializes the checkpointing algorithm. It takes a permanent checkpoint $C_{2,1}$, increments its checkpoint sequence number which is initially zero, and sends checkpoint requests with the checkpoint sequence number $< C_P, 1 >$ to its respective predecessor and successor processes P_1 and P_3. Process P_2 then continues normal computation associated with the application program. Eventually, the checkpoint requests reach their destinations P_1 and P_3. When process P_1 receives the checkpoint request, it first takes a permanent checkpoint $C_{1,1}$ and then increments its checkpoint sequence number and forwards the checkpoint request to its adjacent process P_0 from which it did not yet receive any control message and continues normal computation. Similarly, when process P_3 receives the checkpoint request from the initiator process P_2, it first takes a permanent checkpoint $C_{3,1}$, then increments its checkpoint sequence number. It then forwards the checkpoint request to process P_4 and then continues normal execution. On receiving the checkpoint request, process P_4 first takes a permanent checkpoint $C_{4,1}$, increments its checkpoint sequence number, forwards the checkpoint request to its adjacent process P_0 and then continues normal execution.

Suppose that process P_0 receives the checkpoint request from P_1 before the checkpoint request from P_4. Process P_0 takes a permanent checkpoint $C_{0,1}$, increments its checkpoint sequence number and forwards the request to its adjacent process P_4 from which it did not yet receive the checkpoint request. After forwarding the request, process P_0 continues normal execution. Eventually, process P_4 receives the checkpoint request sent by process P_0. Process P_4 checks its checkpoint sequence number and finds that it has already taken a checkpoint during the current execution of the algorithm and so it discards the request message. Similarly, when process

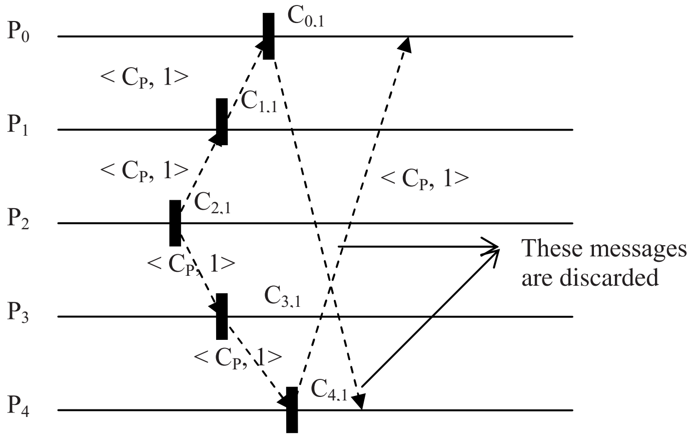

Figure 2. An example of our checkpointing approach.

P_0 receives the checkpoint request sent by process P_4, it checks its checkpoint sequence number and concludes that it has already participated in the current execution of the algorithm and so it discards the request and continues normal execution. It may be observed that in our approach the number of control messages needed is much less than the one in [4]. An estimate of this number is given later.

2.4. Process Behavior

The following discussion is important for designing the checkpointing algorithm for bidirectional ring networks. In any checkpointing algorithm, when a process receives a checkpoint request from an initiator, it may take one of the following two actions:

- either takes a checkpoint by giving the highest priority to the interrupt caused by the checkpoint request, or
- does not respond to the initiator's request (which means that the process is executing a procedure of higher priority and does not take a checkpoint).

In the proposed algorithm, when a process receives a checkpoint request from the initiator, it may decide not to take a checkpoint immediately because it is executing a higher priority procedure. In this case, instead of discarding the checkpoint request to take the checkpoint, the process takes the checkpoint after the conclusion of its current procedure. Any application messages received will be queued up and executed in first in first out manner when the current procedure is completed. Figure 3 illustrates this approach.

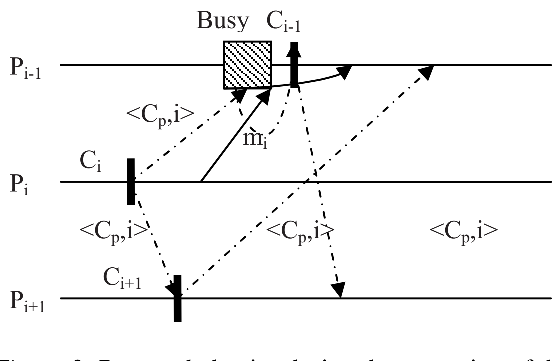

Figure 3. Process behavior during the execution of the checkpointing algorithm.

In Figure 3, assume that it is the kth execution of the checkpointing algorithm and process P_i is the initiator. Process P_i takes a checkpoint C_i, increments its checkpoint sequence number and then sends checkpoint requests to its adjacent processes P_{i-1} and P_{i+1}. Suppose that process P_{i-1} is executing some high priority job and decides not to take a checkpoint. So the checkpoint request is saved in P_{i-1}'s local queue. Assume that process P_i sends an application message m_i after taking the checkpoint C_i to P_{i-1}. The message m_i is also saved in its queue as the process P_{i-1} is busy. After completion of the procedure, process P_{i-1} checks its queue and first processes the checkpoint request and then processes the application message m_i (i.e. following the FIFO order). Therefore, no such application message can ever be an orphan and checkpoints taken during the execution of the algorithm are all mutually consistent. As a result, all checkpoints taken during any kth execution of the checkpointing algorithm are always mutually consistent and, therefore, they are globally consistent checkpoints.

2.4.1. Assumptions

In this work, we assume that the events of receiving the first checkpoint request by a process, taking a checkpoint, incrementing its checkpoint sequence number, and then forwarding request messages to its adjacent processes are done automatically, if the process is not busy with a higher priority job when the checkpoint request arrives. On the other hand, if the process is busy with a higher priority job when the checkpoint request arrives, the events of taking its checkpoint, incrementing its checkpoint sequence number and then forwarding request messages to its adjacent processes are done automatically. We also assume that if a failure occurs during the current execution of the checkpointing algorithm, its execution is immediately aborted and in such a situation, if a process, say P_i has taken its recent permanent ith checkpoint during the current execution of the algorithm, this checkpoint will be discarded and the previous $(i-1)^{th}$ checkpoints of all processes will be considered as the recent GCCs from which the respective processes will restart their normal computation after the system has recovered from the failure.

2.5. Checkpointing Algorithm for Bidirectional Ring Network: Algorithm A

The responsibilities of the initiator process and of all other processes are stated below.

```
Initiator process $P_i$

Step 1: take a checkpoint;
Step 2: increment the checkpoint sequence number;
Step 3: send a checkpoint request $< C_p, i >$ to successor
        and predecessor;
Step 4: continue with normal execution;
        /*Algorithm is terminated at the initiator process*/

At process $P_j$

  it receives the checkpoint request $< C_p, i >$
    if checkpoint request $< C_p, i >$ is the second request received
          /*A process may receive at most two requests from its adjacent
            processes during an execution of the algorithm*/
      discard the request;
      continue normal execution;
    else, if serving a higher priority procedure
      take a checkpoint after the procedure ends;
      increment the checkpoint sequence number;
      forward the checkpoint request $< C_p, i >$ to adjacent process
      from which checkpoint request was not received;
      continue normal execution;
          /* The responsibility of $P_j$ associated with the current
             execution of the algorithm ends*/
    else
      take a checkpoint;
      increment the checkpoint sequence number;
      forward the checkpoint request $< C_p, i >$ to adjacent process
      from which checkpoint request is not received;
      continue normal execution;
          /* The responsibility of $P_j$ associated with the current
             execution of the algorithm ends*/
```

Correctness Proof: We will prove that there cannot exist any orphan message between any two recent checkpoints taken respectively by any two processes during the execution of the checkpointing algorithm.

Consider the following two possible situations that may occur whenever an application message m_i is sent by process P_i to another process P_j;

(1) the application message m_i is sent before P_i sends a checkpoint request to P_j;

(2) the application message m_i is sent after P_i has sent a checkpoint request to P_j;

In the first situation, since after sending the application message m_i process P_i has sent a checkpoint request, it means that process P_i has either initiated the checkpointing algorithm or P_i has received a checkpoint request from another process. According to the algorithm, if process P_i initiates the checkpointing algorithm, it first takes a permanent checkpoint and then sends a checkpoint request to its adjacent processes. On the other hand, if process P_i has received a checkpoint request from another process, it first takes a checkpoint and then forwards the checkpoint request. Hence, in any case, process P_i takes a checkpoint first, before sending a checkpoint request message. Therefore, the application message m_i sent by P_i before it sends its checkpoint request cannot be an orphan.

In the second situation, the application message m_i is sent after P_i has sent the checkpoint request to P_j. Since the communication channel is first in first out, therefore P_j first takes its checkpoint on receiving the checkpoint request and then processes the application message m_i. Therefore, the application message m_i cannot be an orphan.

The above argument is true for all processes in the system. Therefore, orphan messages do not exist in the system. So the checkpoints taken during the execution of Algorithm A are mutually consistent. Hence, the proof follows.

3. Recovery in Bidirectional Ring Network

The proposed recovery approach in this work is explained with an example below. Consider a distributed system of five processes P_0, P_1, P_2, P_3, and P_4 as shown in Figure 4.

Suppose that the checkpointing algorithm has been executed and a set of globally consistent checkpoints is found. Let $\{C_{0,1}, C_{1,1}, C_{2,1}, C_{3,1},$ and $C_{4,1}\}$ be the set of globally consistent checkpoints. Assume that at time t process P_2 fails.

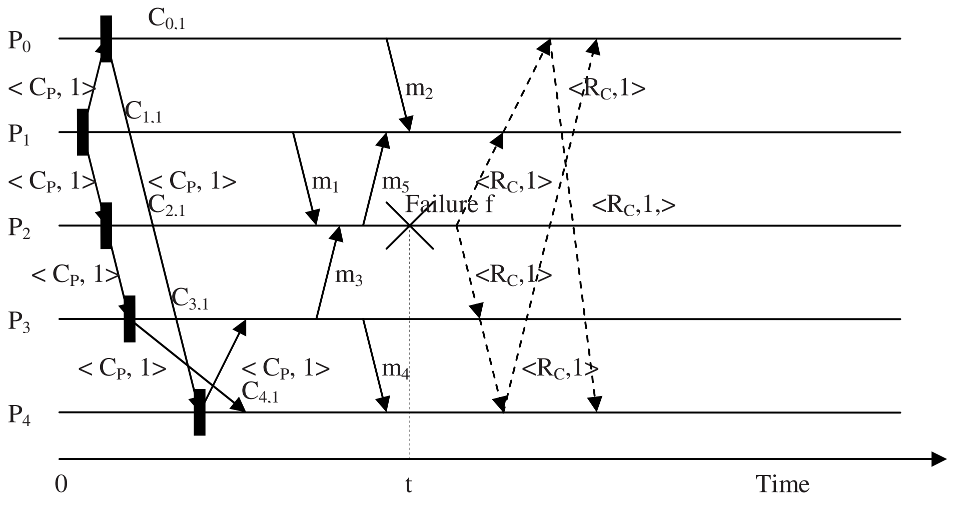

Figure 4. An example of our recovery approach.

After recovering from the failure, P_2 starts the recovery algorithm. Process P_2 sends to its adjacent processes P_1 and P_3 a recovery message $< R_c, 1 >$, where 1 is the checkpoint sequence number to which each receiving process shall roll back. Process P_2then rolls back to its permanent checkpoint $C_{2,1}$ and restarts its normal execution.

On receiving the recovery message $< R_c, 1 >$ process P_1 forwards the recovery message to its predecessor process P_0 from which it did not receive the recovery message and then rolls back to its permanent checkpoint $C_{1,1}$ as specified in the recovery message and restarts computation from that checkpoint. Similarly, process P_3, on receiving the recovery message, forwards the message to its successor process P_4 from which it did not receive the recovery message and rolls back to the checkpoint $C_{3,1}$ and continues normal execution.

Process P_0, on receiving the recovery message from process P_1, forwards the message to P_4 and rolls back to $C_{0,1}$ and continues normal execution. Without any loss of generality, suppose that process P_4 first receives the recovery message sent by P_3. On receiving the recovery message, it first forwards the recovery message to its adjacent process P_0, the one from which it did not yet receive the recovery message, and rolls back to its latest checkpoint $C_{4,1}$ and restarts computation from that checkpoint. Process P_0 discards this recovery message, seeing that this message has the same checkpoint sequence number as the previous one it has received. Process P_4 does the same when it receives the recovery message from P_0.

Observe that, in our approach, as soon as a process receives the first recovery message, it immediately rolls back to its latest consistent checkpoint and restarts computation. But, in [4], restarting at a process is delayed until its *flag_resume* flag becomes true. Also note that our proposed approach uses much less number of control messages.

In brief, the presented recovery approach works as follows. Whenever a failure occurs in a process, after recovering from the failure, the failed process initiates the recovery algorithm. When a process P_i initiates the recovery algorithm, it sends recovery messages to its predecessor and successor processes, say P_k and P_m, with its checkpoint sequence number, and rolls back to its checkpoint taken during the last execution of the checkpointing algorithm and restarts its normal execution. On receiving the recovery message, each process P_j forwards the message to the process other than the one from which it has received the recovery message and then rolls back to its checkpoint as specified in the recovery message and restarts normal execution. Whenever a process receives more than one recovery message (with the same checkpoint sequence number), it discards the later message. Observe that as soon as a process rolls back, its responsibility associated with the recovery algorithm is terminated.

3.1. Recovery Algorithm for Bidirectional Ring Network: Algorithm B

The responsibilities of the initiator process and of all other processes are stated below.

```
Initiator process $P_i$

Step 1: send recovery message $< R_c, i >$ with the current
        checkpoint sequence number j to adjacent processes;
Step 2: rollback to the latest checkpoint;
Step 3: restart from the checkpoint and continue execution;
            /* The Algorithm is terminated at the initiator process */

Any process $P_j$

  it receives the recovery message $< R_c, i >$
    if the recovery message $< R_c, i >$ is the second recovery message received
            /*A process may receive at most two recovery messages from its adjacent processes
              during an execution of the algorithm*/
      discard the request;
      continue execution;
    else
      forward the recovery message $< R_c, i >$ to the adjacent process from which the recovery message was not received;
      rollback to the checkpoint as specified in the recovery message;
      restart from the checkpoint and continue execution;
            /* The responsibility of the process associated with the execution of the recovery
              algorithm is terminated */
```

3.2. Performance

3.2.1. Comparison of the Number of Control Messages Used

We start with a comparison of the number of control messages (i.e. the checkpoint request and recovery messages) used by our proposed checkpointing algorithm and the algorithm reported in [4]. We consider single process initiation of the algorithm in [4]. In the bidirectional ring network, the algorithm proposed in [4] requires two checkpoint requests for a process to take a permanent checkpoint. For the first request each process takes a temporary checkpoint and after receiving the second request the process converts the temporary checkpoint to a permanent checkpoint, and when the initiator receives two requests it terminates the algorithm. Hence, in an n-process system, for the checkpointing algorithm to complete, 2n number of control messages are needed. With the algorithm presented in this paper each process requires only one checkpoint request to take a permanent checkpoint. Therefore, the number of control messages in our proposed approach for a bidirectional ring network is only $n + 1$ in the worst case, which is almost half of the number used in the algorithm proposed in [4]. The recovery algorithm in [4] requires $2n - 1$ control messages. In our presented algorithm, this number is only $n + 1$. It also shows that in our work processes are interrupted much less frequently because of the less number of control messages used. It definitely contributes to the speed of execution of the approaches presented in this work.

3.2.2. Comparison of the Execution Times

The main advantage of our presented checkpointing algorithm is that it creates a globally consistent set of checkpoints in much less time, when compared to the algorithm proposed in [4]. Suppose that t is the average time a message takes to travel from one process to another in an n process system. The checkpointing algorithm in [4] requires n*t time units to complete, whereas our presented checkpointing algorithm requires only (n/2+1)*t time units to complete in the worst case. A summary of the performance comparison is given in Table 1. Observe that we use the same topology as in [4], and so it is correct to assume the existence of the same message passing mechanism in the system, irrespective of what checkpointing algorithm is used. So, we can assume the same value for 't' for both our work and the work in [4].

Algorithm	Property	Mandal (4)	Our Algorithm
Checkpointing Algorithm	Number of Control Messages	2n	n+1
Recovery Algorithm	Number of Control Messages	2n − 1	n+1
Checkpointing Algorithm	Execution Time	n*t	(n/2+1)*t

Table 1. Performance Comparison.

Figure 5a shows the variation of the execution times in the two checkpointing algorithms with the number of processes in the system and Figure 5b shows the variation of the number of control messages used by the two checkpointing algorithms with the number of processes in the system. It is evident that our proposed checkpointing algorithm outperforms the algorithm proposed in [4].

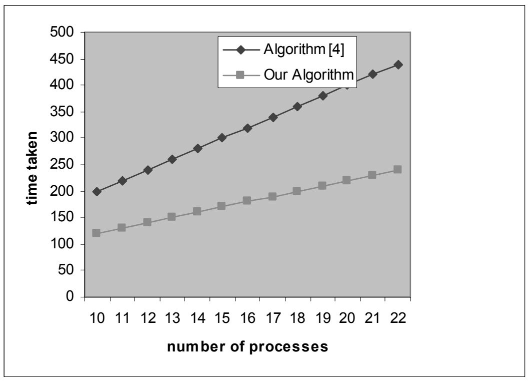

Figure 5a. Execution time vs. no. of processes.

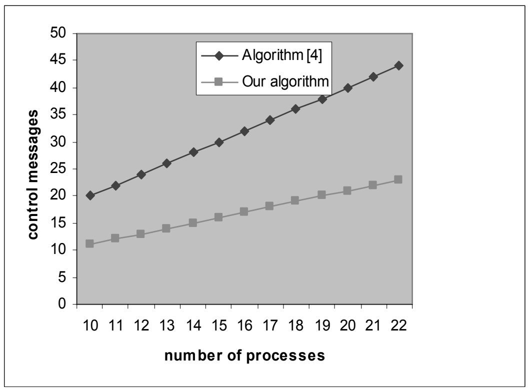

Figure 5b. No. of control messages vs. no. of processes.

In our work, we have not considered concurrent initiations of either the checkpointing or the recovery algorithm. We strongly argue that such initiations create basically message storm in distributed systems, particularly when the number of processes n is large. So this may not be a good practical approach for large n. It is evident that in [4] and [10] the respective message complexities are $O(\mathrm{n}^2)$ and $O(\mathrm{n}^3)$ for concurrent initiations as opposed to $O(\mathrm{n})$ for single initiation in our work.

4. Further Modification

The presented algorithm can be further modified to allow only minimum number of processes to take checkpoints. Note that this minimality consideration has not been addressed in [4]. The modification can be done in the following way.

It is clear that a process does not need to take a checkpoint if it has not sent any application message to its adjacent processes. Suppose that each process maintains a flag, *message_sent*, which is initially set to false. The flag is set to be true the first time a process sends an application message to its adjacent process since its last checkpoint. Every time the process takes a checkpoint, the flag is reset to false. Whenever a process receives a checkpoint request, it checks its *message_sent* flag. If it is true, it takes a checkpoint, increments its checkpoint sequence number, and forwards the message to its adjacent process. If the *message_sent* flag is false, the process does not take a checkpoint and forwards the message to its adjacent process. It can be observed that even if a process does not take a checkpoint in the current checkpointing interval, the latest checkpoints of all the processes still form a consistent global state (CGS) of the system.

5. Conclusion

In this work, we have deviated from the conventional approach of taking temporary checkpoints in the first step and then converting them to permanent ones by processes (as followed by any coordinated checkpointing scheme). The logic for such a deviation has been clearly stated in the problem formulation. It has helped in designing the proposed high performance non-blocking checkpointing and recovery algorithms for bidirectional ring networks. Both our proposed algorithms offer much better performance than the corresponding algorithms reported recently in [4] in terms of the number of control messages used and the execution times of the algorithms. Besides, our checkpointing algorithm can easily be enhanced to allow only the minimum number of processes to take checkpoints, which effectively contributes to the speed of execution of the algorithm, and hence the application program as well. Observe that such minimality consideration has not been addressed in [4]. It may be noted that the checkpointing algorithm in its proposed form cannot be applied to a generic network. It needs major modification for that purpose, which may result in the loss of its efficiency. The reason for this is as follows: in bidirectional networks, a process may receive orphan messages from its predecessor and its successor only; whereas in a generic network, a process may receive orphan messages from any number of the other processes based on the topology. So making checkpoints mutually consistent is a lot more complex problem than the one in ring networks and, therefore, it needs major modification of the proposed checkpointing algorithm to become suitable for arbitrary topology.

References

[1] Y.-M. Wang, Consistent Global Checkpoints that Contain a Given Set of Local Checkpoints. *IEEE Transactions on Computers*, **46**(4) (1997), 456–468.

[2] M. Singhal, N. G. Shivaratri, *Advanced Concepts in Operating Systems.* McGraw-Hill, 1994.

[3] R. Koo, S. Toueg, Checkpointing and Rollback-recovery for Distributed Systems. *IEEE Transactions on Software Engineering*, SE – 13 **1** (1987), 23–31.

[4] P. S. Mandal, K. Mukhopadhyaya, Concurrent Checkpoint Initiation and Recovery Algorithms on Asynchronous Ring Networks. *Journal of Parallel and Distributed Computing*, **64**, Issue 5 (2004), 649–661.

[5] G. Cao, M. Singhal, On Coordinated Checkpointing in Distributed Systems. *IEEE Transactions on Parallel and Distributed Systems*, **9**(12) (1998), 1213–1225.

[6] D. Manivannan, M. Singhal, Quasi-synchronous Checkpointing: Models, Characterization, and Classification. *IEEE Transactions on Parallel and Distributed Systems*, **10** (1999), 703–713.

[7] G. Cao, M. Singhal, Mutable Checkpoints: A New Checkpointing Approach for Mobile Computing Systems. *IEEE Transactions on Parallel and Distributed Systems*, **12** (2001), 157–172.

[8] E. N. Elnozahy, D. B. Johnson, W. Zwaenepoel, The Performance of Consistent Checkpointing. *Proceedings of the 11th Symposium on Reliable Distributed Systems*, (1992), 86–95.

[9] L. M. Silva, J. G. Silva, Global Checkpointing for Distributed Programs. *Proceedings of the 11th Symposium on Reliable Distributed Systems*, (1992), 155–162.

[10] R. Prakash, M. Singhal, Maximum Global Snapshot with Concurrent Initiators. *Proceedings of the 6th IEEE Symposium of Parallel and Distributed Processing*, (1994), 334–351.

[11] M. Jonsson, Two Fibre-ribbon Ring Networks for Parallel and Distributed Computing Systems. *Opt. Eng.*, **37**(12) (1998), 3196–3204.

[12] B. Gupta, N. Mogharreban, S. Rahimi, A. Vemuri, A High Performance Non-blocking Checkpointing / Recovery Algorithm for Ring Networks. *Proceedings of the 2006 Intl. Conf. Parallel and Distributed Processing Techniques and Applications (PDPTA-06)*, (2006) Las Vegas, 234–240.

Received: August, 2006
Recvised: March, 2007
Accepted: July, 2007

Contact addresses:

Bidyut Gupta
Department of Computer Science
Southern Illinois University Carbondale
1000 Faner Drive
Carbondale IL 62901-4511, USA

Shahram Rahimi
Department of Computer Science
Southern Illinois University Carbondale
1000 Faner Drive
Carbondale IL 62901-4511, USA

Ziping Liu
Department of Computer Science
Southeast Missouri State University
One University Plaza
Cape Girardeau, MO 63701-5950, USA

Bidyut Gupta received his M.Tech degree in Electronics Engineering and PhD degree in Computer Science from the University of Calcutta, India in 1978 and 1986, respectively. From 1986 to 1988, he was with the Computer Science Department of Colorado State University at Fort Collins as a visiting member of the faculty. In 1988, he joined the Computer Science Department of Southern Illinois University at Carbondale as assistant professor, where at present he is a full professor and the graduate program director. His research interests include distributed systems, mobile computing, mobile ad-hoc networks, and software agents. He is a senior member of the IEEE.

Shahram Rahimi received his PhD in Scientific Computing from the University of Southern Mississippi in 2002, and his BS degree from National University of Iran (Tehran) in 1992. Currently, he is an associate professor and the undergraduate program director at the Computer Science Department of Southern Illinois University. Dr Rahimi is also the Editor-in-Chief of the International Journal of Computational Intelligence Theory and Practice. His research interests include distributed systems, mobile computing, mobile ad-hoc networks, software agents and soft computing.

Ziping Liu received her M.S. degree in Computer Science and Ph.D. degree in Engineering Science from Southern Illinois University at Carbondale. She is currently an associate professor of Computer Science at Southeast Missouri State University. Her research interests include resource and mobility management of wireless networks, QoS provisioning, mobile ad hoc network and sensor network, design and analysis of algorithms for distributed systems, and programming languages. Before she joined Southeast Missouri State University, she had worked as a software engineer at PCS Motorola.