Novel low-overhead roll-forward recovery scheme for distributed systems

B. Gupta, S. Rahimi and Z. Liu

Abstract: An efficient roll-forward checkpointing/recovery scheme for distributed systems has been presented. This work is an improvement of our earlier work. The use of the concept of forced checkpoints helps to design a single phase non-blocking algorithm to find consistent global checkpoints. It offers the main advantages of both the synchronous and the asynchronous approaches, that is simple recovery and simple way to create checkpoints. The algorithm produces reduced number of checkpoints. Since each process independently takes its decision whether to take a forced checkpoint or not, it makes the algorithm simple, fast and efficient. The proposed work offers better performance than some noted existing works. Besides, the advantages stated above also ensure that the algorithm can work efficiently in mobile computing environment.

1 Introduction

Checkpointing/rollback-recovery strategy provides fault-tolerance to distributed applications [1–5]. A checkpoint is a snapshot of the local state of a process, saved on local non-volatile 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 C is defined as a consistent global checkpoint if no message is sent after a checkpoint of C and received before another checkpoint of C [1]. The checkpoints belonging to a consistent global checkpoint are called globally consistent checkpoints (GCCs).

The concept of roll-forward checkpointing [5–7] is considered to achieve a simple recovery comparable to that in the synchronous approach. This concept helps in limiting the amount of rollback of a process (known as domino effect) in the event of a failure. In this context, it may be noted that in [8] a checkpointing algorithm has been proposed in which processes take checkpoints asynchronously; however all processes have identical time periods to take checkpoints. The work claims to be free from any domino effect. However, we think that this work is more of a synchronous approach than an asynchronous approach; the reason is that checkpoint sequence numbers are used so that all the ith checkpoints of all processes are taken logically at the same time. Hence, the question of domino effect should not arise.

The present work is a modification of the work reported in [7]. The roll-forward checkpointing approach of [7] has been chosen as the basis of the present work because of its simplicity and some important advantages it offers from the viewpoints of both checkpointing and recovery.

doi:10.1049/iet-cdt:20060102

Paper first received 11th July and in revised form 8th December 2006

B. Gupta and S. Rahimi are with the Department of Computer Science, Southern Illinois University, Carbondale, IL 62901-4511, USA

Z. Liu is with the Department of Computer Science, South East Missouri State University, Cape Girardeau, MO 63701, USA

E-mail: bidyut@cs.siu.edu

For a clear understanding of the modifications proposed in the present work, it is required to know clearly how the algorithm in [7] works. For that purpose, in the next section we have stated first its working principle followed by a brief description of its implementation.

2 Related work

The objective of the algorithm in [7] is to design a checkpointing/recovery algorithm that will limit the effect of the domino phenomenon in a distributed computation while at the same time will offer a recovery mechanism that is as simple as in the synchronous checkpointing approach. In order to achieve its goal, in [7] processes go on taking checkpoints (basic checkpoints) asynchronously whereas the roll-forward checkpointing algorithm runs periodically (say the time period is T) by an initiator process to determine the GCCs. During the execution of the algorithm an application process P is forced to take a checkpoint [6, 7] if it has sent an application message m after its latest basic checkpoint. It means that the message m cannot remain an orphan because of the presence of the forced checkpoint. It implies that in the event of a failure occurring in the distributed system before the next periodic execution of the algorithm, process P can restart simply from this forced checkpoint after the system recovers from the failure. However, if process P has not sent any message after its latest basic checkpoint, the algorithm does not force the process to take a checkpoint. In such a situation process P can restart simply from its latest basic checkpoint. In either situation, it has been proven that a process always restarts from its latest GCC as determined by the latest execution of the algorithm and therefore the recovery mechanism is as simple as in the synchronous checkpointing approach. Also it has been shown that the maximum rollback of a process because of a possible domino effect is limited by the time period T, used for the periodic execution of the algorithm. The work has shown that if the concept of taking forced checkpoint is not used, a process may have to rollback substantially compared to when forced checkpoints are used. Thus it achieves roll-forward by limiting the amount of rollback by using both forced checkpoints and periodic execution of the algorithm.

We now give a clear and brief idea about how the algorithm has been implemented using the different data structures used in the algorithm.

Assume that the distributed system has n processes. At its x-th checkpoint, denoted as C_i^x each process P_i maintains an integer vector $\boldsymbol{A}_i^x$ with n elements initialised to zero. $\boldsymbol{A}_i^x[j]$ $(0 \leq j \leq n-1,\ j \neq i)$ denotes the number of messages sent by P_j, and received by P_i. Process P_i increments $\boldsymbol{A}_i^x[j]$ by 1 whenever it receives a message from P_j. The element $\boldsymbol{A}_i[i]$ denotes the total number of messages that process P_i has sent to all other processes and is incremented by one each time when process P_i sends a message. In this work, a checkpoint C_i^x together with the corresponding vector, $\boldsymbol{A}_i^x$, is stored in the stable storage of the distributed system if the checkpoint C_i^x belongs to the set of the GCCs; otherwise in the disk unit of the processor running the process P_i. In other words, processes take checkpoints in their respective disk units and only those checkpoints that are identified by the algorithm as GCCs are copied from the disk units into the stable storage. The reason for this is that since it takes more time to store in stable storage than in disk unit; therefore there is no point in storing a checkpoint in stable storage without knowing if it is a GCC. Otherwise, it will waste time.

In the system, an initiator process maintains an integer vector sum with n elements. Sum$[j]$ $(0 \leq j \leq n-1)$ denotes the total number of messages sent by process P_j, which have been received already by all other processes. The initiator process executes the checkpointing algorithm to determine the GCCs periodically. In its each iteration, it first requests all processes to send their respective $\boldsymbol{A}_i^x$ vectors (i.e. $0 \leq i \leq n-1$) that are stored at the latest respective checkpoints C_i^x $(0 \leq i \leq n-1)$. Second, all processes in turn send their respective vectors to the initiator process. Third, the initiator process stores the vectors $\boldsymbol{A}_i^x$ into a two-dimensional array Store[ ][ ] with $n \times n$ elements such that the ith row of the array Store[ ][ ] contains the vector $\boldsymbol{A}_i^x$. It then sets Store[i][i] $= 0$ for $0 \leq i \leq n-1$. The initiator process then updates Sum$[i]$ for each P_i as Sum$[i] = \Sigma$ Store$[k][i]$, $(0 \leq k \leq n-1)$. This updated Sum$[i]$ denotes the total number of messages received by all other processes so far from process P_i. Then the initiator determines if $\boldsymbol{A}_i^x[i]=$ Sum$[i]$ $-$ d ($d > 0$, i.e. $\boldsymbol{A}_i^x[i] <$ Sum$[i]$). If it is, process P_i has sent d orphan messages after its latest checkpoint. The initiator then asks process P_i to take a checkpoint so that these d messages cannot remain orphan any more. This kind of checkpoints has been termed as forced checkpoints. Therefore in an iteration of the algorithm control messages are exchanged three times between the initiator process and the rest of the processes and hence it is a three-phase algorithm. The algorithm may have to iterate its execution to find whether because of some newly created forced checkpoints, any other process P_k that has not yet taken such a forced checkpoint, needs to do that. However, it ensures that if a process P_k has not sent any message after taking its latest checkpoint (basic), it will not take any forced checkpoint independent of what other processes are doing.

The important advantages of this approach are that effect of the domino phenomenon is limited by the time interval between successive invocations of the algorithm and recovery is as simple as in the synchronous approach. Also at any time, local disk unit of each process stores only one checkpoint per process and same is true for stable storage as well.

The main limitations of this work are as follows: it is a three-phase algorithm that requires $3n$ control messages per iteration. It makes the algorithm much slow. Besides, in its worst case it has to iterate n times resulting in a very large number of the control messages $(=3n^2)$, which in turn generates a large number of interrupts to the processes, thereby slowing down its execution further.

2.1 Problem formulation

The objective of our proposed work is to design a checkpointing/recovery algorithm that will limit the effect of the domino phenomenon in the distributed system while at the same time will offer a recovery mechanism that is as simple as in the synchronous approach. So, effectively our work will follow the basic idea used in [7], that is, processes will go on taking checkpoints (basic) asynchronously, whereas the roll-forward checkpointing algorithm will run periodically to determine the GCCs. However, we will differ from the algorithm in [7] in that our proposed algorithm will aim at making it both single phase and non-blocking so that the algorithm will always terminate in one iteration while using much smaller number of control messages and much less amount of computation per process compared to the same in [7]; this in turn will help in the reduction of the execution time by a good extent when compared to the algorithm of [7]. The key concept used to achieve the objective is that it is the sending process that will make sure that none of its sent messages will remain an orphan. So any process receiving some messages will have no responsibility to make the received messages non-orphan unlike in [7]. This will result in all processes taking their respective check pointing decisions independently and simultaneously without the need for sharing of any information.

Some important and relevant observations are stated in Section 3. The significance of using forced checkpoints is explained in Section 4. In Section 5, we have presented the non-blocking algorithm along with a comparison of the proposed algorithm and the one in [7]. We have given a comparison of our algorithm with some important existing algorithms as well. In Section 6, we have discussed the suitability of our algorithm for mobile computing environment. We have drawn the conclusion in Section 7.

3 Creation of checkpoints and observations

In this work, we have considered the following system model [2, 3]. Processes do not share memory and communicate via messages sent through channels. Channels can lose messages; however, they are made lossless and order of the messages is preserved by some end-to-end transmission protocol. Message sequence numbers may be used to preserve the order. News of a processor failure reaches all other processors in finite time.

3.1 Creation of checkpoints

Assume that the distributed system has n processes (P_0, P_1, ..., P_i, ..., P_{n-1}). Let C_i^x $(0 \leq i \leq n-1,\ x \geq 0)$ denote the x-th checkpoint of process P_i, where i is the process identifier, and x is the checkpoint number. Each process P_i maintains a flag c_i (Boolean). The flag is initially set at zero. It is set at 1 only when process P_i sends its first application message after its latest checkpoint. It is reset to 0 again after process P_i takes a checkpoint. Flag c_i is stored in local RAM of the processor running process P_i for its faster updating. Note that the flag c_i is set to 1 only once independent of how many messages process P_i sends after its latest checkpoint. In addition, process P_i maintains an integer variable N_i which is initially set at 0 and is incremented by 1 each time the algorithm is invoked.

As in the classical synchronous approach [3], we assume that besides the system of n application processes, there exists an initiator process P_I that invokes the execution of the algorithm to determine the GCCs periodically. However, we have shown later that the proposed algorithm can easily be modified so that the application processes can assume the role of the initiator process in turn.

We assume that a checkpoint C_i^x will be stored in stable storage if it is a GCC; otherwise in the disk unit of the processor running the process P_i replacing its previous checkpoint C_i^{x-1}. We have shown that the proposed algorithm considers only the recent checkpoints of the processes to determine a consistent global checkpoint of the system.

We assume that the initiator process P_I broadcasts a control message M_{ask} to all processes asking them to take their respective checkpoints. The time between successive invocations of the algorithm is assumed to be much larger than the individual time periods of the application processes used to take their basic checkpoints.

In this work, unless otherwise specified by ‘a process’ we mean an application (computing) process.

Example 1: Consider the system shown in Fig. 1. Examine the diagram (left of the dotted line). At the starting states of the processes P_0 and P_1, the flags c_0 and c_1 are initialised to zero. The flag c_1 is set at 1 when process P_1 decides to send the message m_1 to P_0. It is reset to 0 when process P_1 takes its basic checkpoint C_1^1. Observe that the flag c_1 is set to 1 only once irrespective of how many messages process P_1 has sent before taking the checkpoint C_1^1. Process P_1 has not sent any message between checkpoints C_1^1 and C_1^2. So, c_1 remains at 0. Also it is clear why c_1 still remains at 0 after the checkpoint C_1^2. Process P_0 sets its flag c_0 to 1 when it decides to send the message m_3 after its latest checkpoint C_0^1.

3.2 Observations

Below, we state some simple but important observations used in the proposed algorithm.

Lemma 1: Consider a system of n processes. If $c_i = 1$, where C_i^k is the latest basic checkpoint of process P_i, then some message(s) sent by P_i to other processes may become orphan.

Proof: The flag c_i is reset to 0 at every checkpoint. It can have the value 1 only between two successive checkpoints of any process P_i if and only if process P_i sends at least one message m between the checkpoints. Therefore $c_i = 1$ means that P_i is yet to take its next checkpoint following C_i^k. Therefore the message (s) sent by P_i after its latest checkpoint C_i^k are not yet recorded. Now if some process P_m receives one or more of these messages sent by P_i and then takes its latest checkpoint before process P_i takes its next checkpoint C_i^{k+1}, then these received messages will become orphan. Hence the proof follows. □

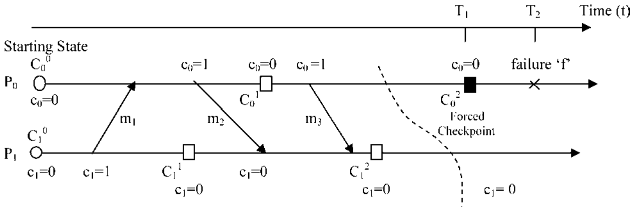

Fig. 1 *Updating of the flags c_0 and c_1*

Lemma 2: If at any given time t, $c_i = 0$ for process P_i with C_i^{k+1} being its latest basic checkpoint, then none of the messages sent by P_i remains an orphan at time t.

Proof: Flag c_i can have the value 1 between two successive checkpoints, say C_i^k and C_i^{k+1}, of a process P_i if and only if process P_i has sent at least one message m between these two checkpoints. It can also be 1 if P_i has sent at least a message after taking its latest checkpoint. It is reset to 0 at each checkpoint. On the other hand, it will have the value 0 either between two successive checkpoints, say C_i^k and C_i^{k+1}, if process P_i has not sent any message between these checkpoints, or P_i has not sent any message after its latest checkpoint. Therefore $c_i = 0$ at time t means either of the following two: (i) $c_i = 0$ at C_i^{k+1} and this checkpoint has been taken at time t. It means that any message m sent by P_i (if any) to any other process P_m between C_i^k and C_i^{k+1} must have been recorded by the sending process P_i at the checkpoint C_i^{k+1}. So the message m cannot be an orphan. (ii) $c_i = 0$ at time t and P_i has taken its latest checkpoint C_i^{k+1} before time t. It means that process P_i has not sent any message after its latest checkpoint C_i^{k+1} till time t. Hence at time t, there does not exist any orphan message sent by P_i after its latest checkpoint. □

Theorem 1: If at any given time t, the set of the latest basic checkpoints, $S = \{C_i^m\}$ $(0 \le i \le n-1)$ and the set of the flags, $S_c = \{c_i | c_i = 0\}$ for all i $(0 \le i \le n-1)$, then the set S consists of n GCCs.

Proof: Without any loss of generality, let us consider a process P_i and examine whether it has sent any message till time t that eventually becomes an orphan. According to Lemma 2, at time t, ‘its flag $c_i = 0$ means that process P_i has not sent any message after its latest checkpoint C_i^m. The same argument holds good for every process in the n process system. Therefore the set S consists of n GCCs. □

4 Significance of forced checkpoints

We illustrate the concept of forced checkpoints from [7]. Our proposed condition of when to take forced checkpoints is different (stated later) and much easier than the one in [7]. Consider the system of Fig. 1 (ignore the checkpoint C_0^2 for the time being). Suppose at time T_2 a failure ‘f’ occurs. According to the asynchronous approach processes P_0 and P_1 will restart their computation from C_0^1 and C_1^1, since these are the latest GCCs.

Now, consider a different approach. Suppose, at time T_1, an attempt is made to determine the GCCs using the idea of forced checkpoints [7]. We start with the recent checkpoints C_0^1 and C_1^2, and find that the message m_3 is an orphan. Observe that the flag c_o of process P_0 is 1, which means that process P_0 has not yet taken a checkpoint after sending the message m_3. However, if at time T_1 process P_0 is forced to take the checkpoint C_0^2 (which is not a basic checkpoint of P_0), this newly created checkpoint C_0^2 becomes consistent with C_1^2. Now, if a failure ‘f’ occurs at time T_2, then after recovery, P_0 and P_1 can simply restart their computation from their respective consistent states C_0^2 and C_1^2. Observe that process P_1 now restarts from C_1^2 in the new situation instead of restarting from C_1^1. Therefore the amount of rollback per process has been reduced. Note that these two latest checkpoints form a recent consistent global checkpoint as in the synchronous approach.

The following condition states when a process has to take a forced checkpoint.

Condition C: For a given set of the latest checkpoints (basic), each from a different process in a distributed system, a process P_i is forced to take a checkpoint C_i^{m+1}, if after its previous checkpoint C_i^m belonging to the set, its flag $c_i = 1$.

Proposition 1: Let C_i^m and C_i^{m+1} be the two consecutive checkpoints of process P_i, such that only C_i^{m+1} is the forced one. Then, a message sent by P_i between these two checkpoints (i.e. between these two checkpoints $c_i = 1$) can never be an orphan. However, it may be a lost message.

We shall now prove that at a given time t the set of the latest n checkpoints (including both the forced ones taken at time t, as well as the basic ones of those processes that do not need to take forced checkpoints at least till time t) is the set of the GCCs at time t. Let the set be S^*.

Theorem 2: The n checkpoints in S^* are globally consistent at time t.

Proof: The following two cases are the only possible cases:

Case 1: There is no forced checkpoint in S^*.

Proof follows directly from Theorem 1.

Case 2: S^* has both basic and forced checkpoints.

Without any loss of generality, consider a process P_i with its latest checkpoint C_i^{m+1} belonging to the set S^* and its flag c_i at time t. The following two possible situations need to be considered.

Let checkpoint C_i^{m+1} be a basic checkpoint. Since process P_i does not need to take a forced checkpoint after its latest basic checkpoint C_i^{m+1} till time t therefore its flag $c_i = 0$ after this checkpoint at least till time t. In other words, it means that process P_i has not sent any message since its latest checkpoint till time t (Lemma 2). Therefore there is no question of having an orphan message sent by process P_i after its latest checkpoint C_i^{m+1} till time t.

Let checkpoint C_i^{m+1} be a forced checkpoint taken by process P_i. Since process P_i has taken the forced checkpoint C_i^{m+1} at time t after its previous checkpoint C_i^m; therefore it means that process P_i has sent at least one message m_i to another process P_j. However, the event of sending the message m_i has been recorded in the forced (latest) checkpoint C_i^{m+1}. Therefore irrespective of whether the message m_i has been received by process P_j or not, the message m_i can never be an orphan.

The above arguments are true for each process with its latest checkpoint belonging to the set S^*. Hence the set S^* represents a consistent global checkpoint of the n-process system at time t. □

5 Non-blocking approach

We explain first the problem associated with non-blocking approach. Consider a system of two processes P_i and P_j. Assume that both processes have sent messages after their last checkpoints. So both c_i and c_j are set at 1. Assume that the initiator process P_I has sent the request message M_{ask}. Let the request reach P_i before P_j. Then P_i takes its checkpoint C_i^k because $c_i = 1$ and sends a message m_i to P_j. Now consider the following scenario.

Suppose a little later process P_j receives m_i and still P_j has not received M_{ask}. So, P_j processes the message. Now the request from P_I arrives at P_j. Process P_j finds that $c_j = 1$. So it takes a checkpoint C_j^r. We find that message m_i has become an orphan because of the checkpoint C_j^r. Hence, C_i^k and C_j^r cannot be consistent.

5.1 Solution to the non-blocking problem

To solve this problem, we propose that a process be allowed to send both piggybacked and non-piggybacked application messages. We explain the idea below.

Each process P_i maintains an integer variable N_i, initially set at 0 and is incremented by 1 each time process P_i receives the message M_{ask} from the initiator. Thus variable N_i represents how many times the check pointing algorithm has been executed including the current one (according to the knowledge of process P_i). Note that at any given time t, for any two processes P_i and P_j, their corresponding variables N_i and N_j may not have the same values. It depends on which process has received the message M_{ask} first. However, it is obvious that $|N_i - N_j|$ is either 0 or 1.

Below we first state the solution for a two-process system. The idea is similarly applicable for an n process system as well.

5.1.1 Two-process solution: Consider a distributed system of two processes P_i and P_j only. Assume that P_i has received M_{ask} from the initiator process P_I for the k th execution of the algorithm, and has taken a decision whether to take a checkpoint or not, and then has implemented its decision. Also assume that P_i now wants to send an application message m_i for the first time to P_j after it finished participating in the k th execution of the algorithm. Observe that P_i has no idea whether P_j has received M_{ask} yet and has taken its checkpoint. To make sure that the message m_i can never be an orphan, P_i piggybacks m_i with the variable N_i. Process P_j receives the piggybacked message $\langle m_i, N_i\rangle$ from P_i. We now explain why message m_i can never been an orphan. Note that $N_i = k$; that is it is the kth execution of the algorithm that process P_i has last been involved with. It means the following to the receiver P_j of this piggybacked message:

(1) Process P_i has already received M_{ask} from the initiator P_I for the k th execution of the algorithm,
(2) P_i has taken a decision if it needs to take a forced checkpoint and has implemented it,
(3) P_i has resumed its normal operation and then has sent this application message m_i.
(4) The sending event of message m_i has not yet been recorded by P_i.

Since the message contains the variable N_i, process P_j compares N_i and N_j to determine if it has to wait to receive the request message M_{ask}. Based on the results of the comparison process P_j takes one of the following two actions, stated below as Observations 1 and 2.

Observation 1: If $N_i\ (=k) > N_j\ (=k-1)$, process P_j now knows that the kth execution of the check pointing algorithm has already begun and so very soon it will also receive the message M_{ask} from the initiator process associated with this execution. So instead of waiting for M_{ask} to arrive, it decides if it needs to take a checkpoint and implements its decision, and then processes the message m_i. After a little while when it receives the message M_{ask} it just ignores it. Therefore message m_i can never be an orphan.

Observation 2: If $N_i = N_j = k$, like process P_i, process P_j also has received already the message M_{ask} associated with the latest execution (kth) of the check pointing algorithm and has taken its check pointing decision and has already implemented that decision. Therefore process P_j now processes the message m_i. It ensures that message m_i can never be an orphan, because both the sending and the receiving events of message m_i have not been recorded by the sender P_i and the receiver P_j, respectively.

Observation 3: Process P_i does no more need to piggyback any application message to P_j till the $(k+1)$th invocation (next) of the algorithm. The reason is that after receiving the piggybacked message $\langle m_i, N_i \rangle$, P_j has already implemented its decision whether to take a checkpoint or not before processing the message m_i. If it has taken a checkpoint, then all messages it receives from P_i starting with the message m_i cannot be orphan. So it processes the received messages. Also if P_j did not need to take a checkpoint during the kth execution of the algorithm, then obviously the messages sent by P_i to P_j staring with the message m_i till the next invocation of the algorithm cannot be orphan. So it processes the messages.

Therefore for an n process distributed system, process P_i piggybacks only its first application message sent (after it has implemented its check pointing decision for the current execution of the algorithm and before its next participation in the algorithm) to process P_j, where $j \neq i$, and $0 \leq j \leq n-1$.

From the above discussion, it is clear that process P_j starts executing its responsibility associated with the algorithm when one of the following two events occurs: (1) P_j has received the request message M_{ask} from the initiator process, and (2) P_j has received a piggybacked application message $\langle m_i, N_i \rangle$ with $N_i > N_j$, but has not yet received the message M_{ask}. Therefore occurrence of the second event means that process P_j tests first if $N_i > N_j$ and finds it true before starting the execution.

5.2 Algorithm non-blocking

Initiator process P_I: it broadcasts M_{ask} to all P_j, for $0 \leq j \leq n-1$ /∗ It is the kth invocation of the algorithm ∗/

Every process P_j executes the algorithm as given in Fig. 2.

```
if Pj receives Mask
      sets Nj = Nj + 1;
      if cj = 1
          Pj sets cj = 0; stores cj in its local storage (RAM);
          takes the forced checkpoint Cj^(x+1) and stores Cj^(x+1) in both stable storage and its disk;
          /* Cj^(x-1) replaces Cj^x in disc unit and it is the latest GCC of Pj */
          Pj resumes computation;
      else
          Pj stores a copy of its last checkpoint (basic) Cj^x from its disc to stable storage;
          / * Cj^x is the latest GCC of Pj */
          Pj resumes computation;
else if   Pj receives < mi, Ni>, for any i ≠ j && Pi has not yet received Mask for the current execution of
      the check pointing procedure                                  /* Ni (= k) > Nj (= k-1) */
      sets Nj = Nj + 1;
      if cj = 1
          Pj sets cj = 0; stores cj in its local storage (RAM);
          takes the forced checkpoint Cj^(x+1) and stores Cj^(x+1) in both stable storage and disk;
          /* Cj^(x+1) replaces Cj^x in disc unit and it is the latest GCC of Pj */
          processes the received message mi;
          continues its normal operation and ignores Mask, when received for the current
          execution of the check pointing procedure;
      else
          Pj stores a copy of its last checkpoint (basic) Cj^x from its disc to stable storage;
          / * Cj^x is the latest GCC of Pj */

          processes any received message mi;
          continues its normal operation and ignores Mask, when received for the current
          execution of the check pointing procedure;
```

Fig. 2 *Execution of process P_j*

```
At each process P_i (0 ≤ i ≤ n-1):
   if CLK_i = (i+(counter_i*n))*T
      counter_i = counter_i+1;                /* P_i becomes the initiator*/
```

Fig. 3 *Seletion of an initiator process*

Proof of correctness: In the first 'if else' block of the pseudo code, each process P_j decides based on the value of its flag c_j whether it needs to take a checkpoint. If it has to take a checkpoint, it resets c_j to 0. Therefore in other words, each process P_j makes sure using the logic of Lemma 2 that none of the messages, if any, it has sent since its last checkpoint can be an orphan. On the other hand, if P_j does not take a checkpoint, it means that it has not sent any message since its previous checkpoint.

In the 'else if' block each process P_j follows the logic of Observations 1, 2 and 3 which ever is appropriate for a particular situation so that any application message (piggy-backed or not) received by P_j before it receives the request message M_{ask} cannot be an orphan. Besides none of its sent messages, if any, since its last checkpoint can be an orphan as well (following the logic of Lemma 2).

Since Lemma 2, and Observations 1, 2 and 3 guarantee that no sent or received message by any process P_j since its previous checkpoint can be an orphan and since it is true for all participating processes; therefore following the logic of Theorems 1 and 2, the algorithm guarantees that the latest checkpoints taken during the current execution of the algorithm and the previous checkpoints (if any) of those processes that did not need to take checkpoints during the current execution of the algorithm are GCCs. □

The above algorithm uses a separate initiator process. However, there is no such need and the algorithm can easily be modified so that the application processes assume the role of the initiator in turn. For selecting the next initiator, each process P_i maintains a local variable CLK_i which is incremented at periodic time interval T (time between successive invocations of the algorithm). Also P_i maintains a local counter, counter_i which is set to 0 initially. P_i increments counter_i during its turn to initiate the algorithm. Using these two variables process P_i determines independently when to initiate. In Fig. 3, we state how process P_i does it.

Observe that initiator P_i broadcasts the message M_{ask} to the rest $(n-1)$ processes before starting its execution of the algorithm and any process P_j, $(j \neq i)$ starts executing the algorithm only after receiving M_{ask}.

5.3 Performance comparison with the algorithm in [7]

We first state the common advantages which are offered by both our approach and the one in [7]. Then we shall state the unique advantages which our algorithm only offers.

5.3.1 Common advantages of the two algorithms: Below we state some important common advantages which both algorithms offer.

(1) At any time, stable storage contains only one checkpoint per process. Also at any time local disk unit of each process contains only one checkpoint. Besides both algorithms allow their processes to keep their data structures in their respective local memory (RAM) for their fast access.
(2) Effect of the domino phenomenon is limited by the time interval between successive invocations of the algorithm and recovery is as simple as in the synchronous approach.
(3) Both algorithms create only those forced checkpoints that are needed.
(4) Creation of useless basic checkpoints (which cannot be GCCs) can be avoided following the scheme of [7].

5.3.2 Unique advantages offered by the proposed algorithm: The fundamental difference between our algorithm and the one in [7] is the condition about when to take a forced checkpoint. It leads to the following advantages and improved complexity when compared to [7].

(1) The proposed algorithm is a single-phase algorithm unlike in [7], that is, all processes can take their respective check pointing decisions independently and simultaneously. Hence our algorithm is expected to be faster.
(2) Ours is a non-blocking approach where as in [7] processes may need to block their underlying computation during the execution of the algorithm.
(3) In the proposed algorithm, each process is interrupted only once (by the message M_{ask}) compared to three in [7]. This makes the algorithm faster.
(4) The data structures needed in the present work are very simple compared to most of the existing works in this area. Each process maintains just a Boolean flag and an integer variable. In [7], each process maintains a vector of length n for an n process system and has to update it each time it sends or receives a message. It causes large number of interrupts to any computing process, resulting in delaying the computation further. In our approach, the flag is at most updated twice between two consecutive checkpoints causing much less interrupts.
(5) Our approach is a single phase one that is, for an n-process distributed system the initiator process broadcasts only once a request message to all other processes, and the rest of the processes, after receiving the respective request messages take their checkpointing decisions independently. Therefore the total number of control (request) messages is always n, so our message complexity is $O(n)$. Note that if we assume the presence of special hardware to facilitate broadcasting, it becomes $O(1)$.
In [7], in its best case, that is when the algorithm terminates only after the first iteration, $3n$ control massages are exchanged between the initiator process and the rest of the processes. So its message complexity in the best case is $O(n)$. However, in its worst case, that is when the algorithm has to iterate n times such that in each iteration only one process uniquely takes a forced checkpoint, the total number of control messages exchanged becomes $3n^2$ resulting in a message complexity of $O(n^2)$. Fig. 4 illustrates the nature of the variation of the number of control messages with the increase in the number of the processes in the system.
(6) In our approach, the algorithm is a single phase one, that is, it terminates in one iteration. When a process P_j receives

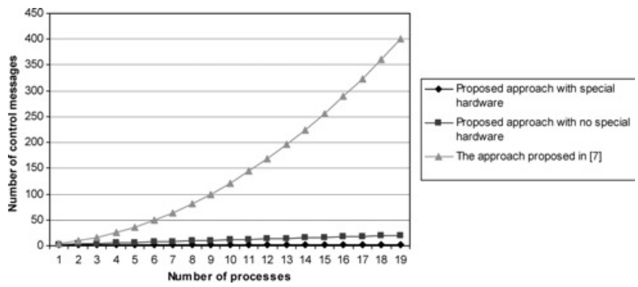

Fig. 4 *Number of control messages against number of processes for the proposed approach, with and without special hardware, and the model in [7]*

Table 1: System performance

	System messages
Koo–Toueg [2]	$3*n_{min}*n_{dep}*C_{air}$
Elnozahy [10]	$2*C_{broad} + n*C_{air}$
Cao–Singhal [9]	$\simeq 2*n_{min}*C_{air} + \min(n_{min}*C_{air}, C_{broad})$
our algorithm	C_{broad}

the request message from the initiator process, it just tests independently if its Boolean flag $c_j = 1$ or 0 in order to decide if it needs to take a forced checkpoint. This is true for all the processes. Hence the computational complexity is $O(n)$. Note that the computational complexity of the algorithm in [7] is $O(n^3)$ for the worst case scenario, whereas ours is always $O(n)$. However, if we assume the presence of special hardware to facilitate broadcasting, it becomes $O(1)$ in our approach.

5.4 Performance comparison with other noted works

The performance of our algorithm is compared with some noted algorithms [2, 9, 10] from the viewpoint of the number of control (system) messages used by each. We use the following notations (and some of the analysis from [9]) for comparison. The analytical comparison is given in the Table 1. In this table:

C_{air}: cost of sending a message from one process to another process,
C_{broad}: cost of broadcasting a message,
n_{min}: number of processes that need to take checkpoints,
n: total number of processes in the system,
n_{dep}: average number of processes on which a process depends.

In Figs. 5 and 6, we illustrate how the number of control messages (system messages) sent and received by processes is affected by the increase in the number of the processes in the system. In Fig. 5, n_{dep} factor is considered being 5% of the total number of processors in the system and C_{broad} is equal to C_{air} (assuming that special hardware is used to facilitate broadcasting – which is not the case most of the times). As Fig. 5 shows, the number of control messages does not increase with the increase in the number of processes in our approach unlike other approaches.

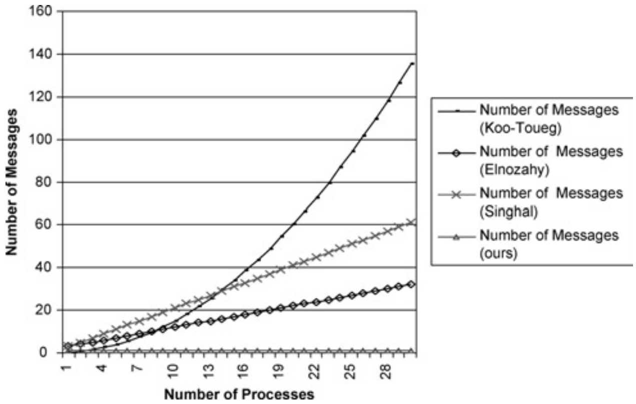

Fig. 5 *Number of messages against number of processes for four different approaches when $C_{broad} = C_{air}$*

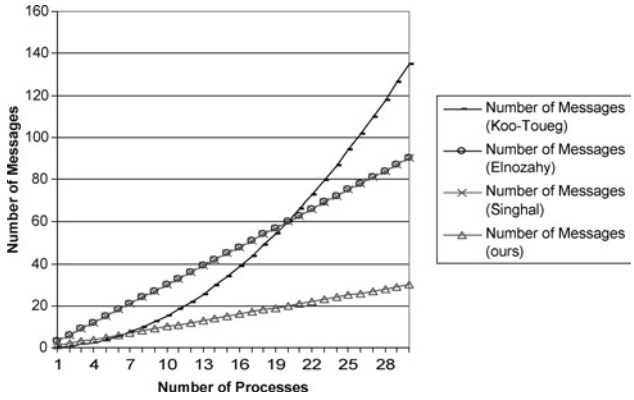

Fig. 6 *Number of messages against number of processes for four different approaches when $C_{broad} = n*C_{air}$*

In Fig. 6, we have considered absence of any special hardware for broadcasting and therefore assumed C_{broad} to be equal to $n * C_{air}$. In this case, although the number of messages does increase in our approach, but it stays smaller compared to other approaches when the number of the processes is higher than 7 (which is the case most of the times).

6 Suitability for mobile computing environment

A distributed algorithm running in a mobile computing environment must offer efficient use of the limited wireless bandwidth needed for communication among the computing processes, the mobile hosts' limited battery power and limited memory. Below we justify that the proposed algorithm satisfies all these three requirements.

- It offers efficient use of the wireless bandwidth, because the algorithm is a single-phase algorithm with only one control message (M_{ask}).
- It offers efficient use of the mobile hosts' battery power, because (1) each mobile host is interrupted only once by the message M_{ask}. It saves time since interrupt handling time cannot be ignored. Note that in other approaches [9, 11, 12] it is more than one, and (2) each process P_i only checks if its $c_i = 1$ to decide if it needs to take a checkpoint. This is the only computation that a mobile host is involved with.
- It offers efficient use of the mobile hosts' memory, because (1) we use very simple data structure: an integer variables N_i and a Boolean variable c_i per process. Note that this amount is much less than the same in [9, 11, 12], and (2) at any time local disk unit of a mobile host stores only one checkpoint.

7 Conclusion

We have presented a single-phase non-blocking check pointing algorithm that ensures simple recovery. The noteworthy point of the presented approach is that a process receiving a message does not need to worry whether the received message may become an orphan or not. It is the responsibility of the sender of the message to make it non-orphan. Because of this, each process is able to perform its responsibility independently and simultaneously with others just by testing its local Boolean flag. This makes the algorithm a single phase one and thereby, in effect, makes the algorithm fast, simple and efficient. Also the computational complexity and the message complexity are both $O(n)$. These advantages along with its 'single-phase non-blocking' nature make the algorithm suitable for mobile computing environment as well.

8 Acknowledgment

We feel grateful to the referees for their valuable suggestions that have helped immensely in preparing the revised manuscript.

9 References

1 Wang, Y.-M.: 'Consistent global checkpoints that contain a given set of local checkpoints', *IEEE Trans. Comput.*, 1997, **46**, (4), pp. 456–468
2 Koo, R., and Toueg, S.: 'Check pointing and rollback-recovery for distributed systems', *IEEE Trans. Software Eng.*, 1987, **13**, (1), pp. 23–31
3 Venkatesan, S., Juang, T.T.-Y., and Alagar, S.: 'Optimistic crash recovery without changing application messages', *IEEE Trans. Parallel Distrib. Syst.*, 1997, **8**, (3), pp. 263–271
4 Cao, G., and Singhal, M.: 'On coordinated check pointing in distributed systems', *IEEE Trans. Parallel Distrib. Syst.*, 1998, **9**, (12), pp. 1213–1225
5 Pradhan, D.K., and Vaidya, N.H.: 'Roll-forward check pointing scheme: a novel fault-tolerant architecture', *IEEE Trans. Comput.*, 1994, **43**, (10), pp. 1163–1174
6 Gass, R.C., and Gupta, B.: 'An efficient check pointing scheme for mobile computing systems'. Proc. ISCA 13th Int. Conf. Computer Applications in Industry and Engineering, Honolulu, USA, November 2000, pp. 323–328
7 Gupta, B., Banerjee, S.K., and Liu, B.: 'Design of new roll-forward recovery approach for distributed systems', *IEE Proc., Comput. Digit. Tech.*, 2002, **149**, (3), pp. 105–112
8 Manivannan, D., and Singhal, M.: 'Asynchronous recovery without using vector timestamps', *J. Parallel Distrib. Comput.*, 2002, **62**, (12), p. 1695—1728
9 Cao, G., and Singhal, M.: 'Mutable checkpoints: a new check pointing approach for mobile computing systems', *IEEE Trans. Parallel Distrib. Syst.*, 2001, **12**, (2), pp. 157–172
10 Elnozahy, E.N., Johnson, D.B., and Zwaenepoel, W.: 'The performance of consistent check pointing'. Proc. 11th Symp. on Reliable Distributed Systems, 1992, pp. 86–95
11 Ahmed, R., and Khaliq, A.: 'A low-overhead check pointing protocol for mobile networks', *IEEE CCECE*, 2003, **3**, pp. 4–7
12 Kumar, P., Kumar, L., Chauhan, R.K., and Gupta, V.K.: 'A non-intrusive minimum process synchronous checkpointing protocol for mobile distributed systems'. ICPWC 2005, Proc. IEEE Int. Conf. Personal Wireless Communications, New Delhi, India, January 2005, pp. 491–495