Design of new roll-forward recovery approach for distributed systems

B. Gupta, S.K. Banerjee and B. Liu

Abstract: A new roll-forward checkpointing scheme is proposed using basic checkpoints. The direct-dependency concept used in the communication-induced checkpointing scheme is applied to basic checkpoints to design a simple algorithm to find a consistent global checkpoint. Both blocking (i.e. when the application processes are suspended during the execution of the algorithm) and non-blocking approaches are presented. The use of the concept of forced checkpoints ensures a small re-execution time after recovery from a failure. The proposed approaches enjoy the main advantages of both the synchronous and the asynchronous approaches, i.e. simple recovery and simple way to create checkpoints. Besides, in the proposed blocking approach, the direct-dependency concept is implemented without piggybacking any extra information with the application message. A very simple scheme for avoiding the creation of useless checkpoints is also proposed.

1 Introduction

The checkpointing/rollback-recovery strategy represents an attractive approach for providing fault-tolerance to distributed applications [1–11]. A checkpoint is a snapshot of the local state of a process, saved in 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 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). A checkpoint C_i of a process P_i is called a useless checkpoint if it does not belong to any consistent global checkpoint of a distributed system.

There are two fundamental approaches for checkpointing and recovery. One is the synchronous approach while the other is the asynchronous approach [3]. The synchronous checkpointing approach assumes that a single process other than the application processes invokes the algorithm periodically to determine a consistent global checkpoint. This process is known as the initiator process. It asks periodically all application processes to take checkpoints in a co-ordinated way. The co-ordination is performed in such a way that the checkpoints taken by the application processes always form a consistent global checkpoint of the system. This co-ordination is actually achieved through the exchange of additional (control) messages. It causes some delay (known as the synchronisation delay) during normal operation. This is the main drawback of this method. However, the main advantage is that the set of checkpoints taken periodically by the different processes always represents a consistent global checkpoint. Thus, after the system recovers from a failure, each process knows where to rollback in order to restart its computation. In fact, the restarting state will always be the recent consistent global checkpoint. Therefore, recovery is very simple. On the other hand, if failures rarely occur between successive checkpoints, then the synchronous approach places an unnecessary burden on the system in the form of additional messages and delay.

In the asynchronous approach, processes take their checkpoints independently. Taking checkpoints is thus very simple as no co-ordination between the processes is required while taking the checkpoints. After a failure occurs, a procedure for rollback-recovery attempts to build a consistent global checkpoint [3]. However, in this approach, because of the absence of any co-ordination between the processes, there may not exist a recent consistent global checkpoint, which may cause a rollback of the computation. This is known as the domino effect. In the worst case of the domino effect, after recovery from a failure all the processes may have to rollback to their respective initial states to restart their computation. Hence, compared to the synchronous approach, recovery is more complex while taking checkpoints is much simpler. Note that the synchronous approach is free from any domino effect.

Besides these two fundamental approaches, there is another approach, known as the communication-induced checkpointing approach. In this approach, processes co-ordinate to take checkpoints by piggybacking some control information on application messages [6, 7]. However, this co-ordination does not guarantee that a recent global checkpoint will be consistent. This means that this approach also suffers from the domino effect. Therefore, a recovery algorithm has to search for a consistent global checkpoint before the processes can restart their computa-

IEE Proceedings online no. 20020410

DOI: 10.1049/ip-cdt:20020410

Paper first received 7th June 2001 and in revised form 8th March 2002

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

S.K. Banerjee is with the Department of Computer Science, University of Calcutta, Calcutta 700009, India

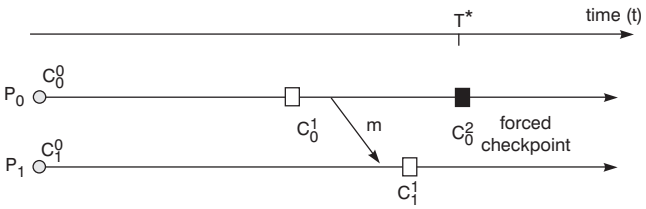

Fig. 1 *Forced checkpoint*

tion after recovery from a failure. In this approach, taking checkpoints is simpler than in the synchronous approach, while the recovery process is more complex.

In this work, we follow the asynchronous approach for taking checkpoints. A process takes its checkpoint independently and periodically. The periods of taking checkpoints by means of distinct processes are not related. In this work, such checkpoints are termed basic checkpoints. We now introduce the concept of forced checkpoints [12, 13] as this kind of checkpoint together with the above-mentioned basic checkpoints will be used in the present work. We explain this concept below, using the example of Fig. 1.

Consider a system comprising two processes P_0 and P_1. The two processes have their initial states saved at the checkpoints C_0^0 and C_1^0, respectively (ignore the checkpoint C_0^2 for the time being). Assume that after the system starts its normal operation, C_0^1 and C_1^1 are the first basic checkpoints taken by the two processes. Suppose that P_0 sends a message m to P_1 after it has taken the checkpoint C_0^1. Suppose at time T* we try to find a consistent global checkpoint of the system. Note that the message m is an orphan message since the event of sending it is not recorded in the checkpoint C_0^1, but the event of receiving the message is recorded in the checkpoint C_1^1. Therefore, the checkpoints C_0^1 and C_1^1 cannot form a consistent global checkpoint. However, the checkpoints C_0^1 and C_1^0 form a consistent global checkpoint as there is no orphan message between them. Note that at this time the checkpoint C_1^1 is a useless checkpoint. Now consider a different scenario, in which at time T* process P_0 is asked to take the checkpoint C_0^2, which is not a basic checkpoint. Message m can no longer be an orphan message with respect to checkpoints C_0^2 and C_1^1. Therefore these two checkpoints form a consistent global checkpoint. In other words, they are now GCCs with respect to each other. The checkpoint C_0^2 is known as the forced checkpoint [12, 13]. The significance of taking such forced checkpoints for recovery purposes in the present work is explained in detail in Section 5.

2 Related work

In this paper, the concept of roll-forward checkpointing [13–16] is used using basic checkpoints in order to ensure simple recovery comparable to that in the synchronous approach. In [14], the concept of roll-forward checkpointing has been used to design the architecture for a fault-tolerant multiprocessor environment, the objective being the achievement of the performance of a triple modular redundant system using duplex system redundancy. In [15], the concept has been used to achieve low-cost checkpointing/recovery for mobile computing systems by piggybacking dependency information (a Boolean vector of size N for an N process system) with each application message. Our work is not about designing any fault-tolerant architecture for multiprocessor systems. It is also different from the work in [15] in that it considers only basic checkpoints, and no piggybacking of dependency information with the application is needed in the case of the blocking approach. In [13], in which a mobile computing environment is considered, it is guaranteed that no piggybacking of any control information is needed. Moreover, the work considers communication-induced checkpoints and assumes only a blocking approach, i.e. application processes are suspended when the algorithm to determine a consistent global checkpoint is executed.

In this work, we have presented both blocking and non-blocking approaches to determine the GCCs. (A preliminary version of this present work, which considers the blocking approach only, has been reported in [16]). In the blocking approach, application processes are suspended during the execution of the algorithm to find the GCCs, while in the later approach, application processes are not suspended. We have used and appropriately modified the idea of the non-blocking (also called non-intrusive) approach as reported in [15] to make it suitable for basic checkpoints. The proposed algorithm runs periodically to determine the GCCs as in the synchronous approach. Thus after a failure, the processes know where to rollback, guaranteeing a very simple recovery. Our approach is preferable to the synchronous approach in that there is no need for the processes to be synchronised while the GCCs are determined (i.e. there is no synchronisation delay). To achieve this result, we apply the direct-dependency concept (used in the communication-induced approaches) to basic checkpoints, and the concept of forced checkpoints [12, 13]. Note that the asynchronous checkpointing approach is considered in this work because it is one of the least troublesome methods for taking checkpoints. The above-mentioned two concepts help in designing a simple algorithm for determining the GCCs. Also, since we take the GCCs periodically, the amount of rollback is limited by the time between successive invocations of the algorithm and thus is comparable to that in the synchronous approach. In other words, it can be said that roll-forward is achieved by limiting the amount of rollback. We also show that in the present work only the processes that have sent some orphan messages will be forced to take checkpoints. In fact, in our worst-case scenario, N processes of an 'N-process system' need to take forced checkpoints, whereas in the synchronous approach all N processes always have to take checkpoints. In other words, our worst case is identical to the synchronous approach.

3 Creation of checkpoints

3.1 System model

The following assumptions are made about the distributed systems considered in this work [3–6].

1. Processes do not share memory and communicate via messages sent through channels.
2. Channels can lose messages. However, they are made virtually lossless and the orders of the messages are preserved by some end-to-end transmission protocol.
3. The news of a processor failure reaches all other processors in finite time.
4. Processes are piecewise deterministic.

3.2 Creation of checkpoints

Assume that the distributed system has N processes (P_0, $P_1, \ldots, P_i, \ldots, P_{N-1}$). Let C_i^x ($0 \leq i \leq N-1, x \geq 0$) denote the xth checkpoint of process P_i, where i is the process

identifier, and x is the checkpoint number. Each process P_i maintains an integer vector A_i with N elements initialised to zero. $A_i[j]$ $(0 \leq j \leq N-1, j \neq i)$ denotes the number of messages sent by P_j, and received by P_i. P_i increments $A_i[j]$ by 1 whenever it receives a message from P_j. Here $A_i[i]$ has a special meaning. It means the total number of messages that process P_i has sent to all other processes, i.e. $A_i[i]$ acts as a message counter of P_i. The entry $A_i[i]$ is also incremented by one each time process P_i sends a message.

We shall denote by A_i^x, the vector stored by process P_i together with its checkpoint C_i^x. Note that the vector A_i immediately prior to taking the checkpoint C_i^x and the vector A_i^x are identical. We shall also use the notation A_i' to denote the last vector created by process P_i after its lastest checkpoint. In this work, we assume that a checkpoint C_i^x together with the corresponding vector will be stored in the stable storage of the distributed system if the checkpoint C_i^x belongs to the set of the GCCs; otherwise it is assumed to be stored 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 the disk unit, there is no point in storing a checkpoint in stable storage without knowing if it is a GCC. Otherwise, it will waste time. Also note that the vectors in between two consecutive checkpoints are stored in the processor's local memory (RAM) for faster updating when the application processes receive or send messages.

In the system an initiator process (different from the application processes) maintains another 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 already been received by all other processes. The initiator process executes the checkpointing algorithm to determine the GCCs periodically. To run the algorithm it collects the A_i^x values from all processes (i.e. $0 \leq i \leq N-1$) that are stored at the latest respective checkpoints C_i^x $(0 \leq i \leq N-1)$. It also collects the respective latest updated vectors A_i' of the processes. We shall not elaborate upon the use of A_i' in this Section, since we shall do so in Section 5. In this Section, we explain how the vectors A_i^x are used to update the vector sum. In this work, unless otherwise specified, by 'a process' we mean an application process.

The initiator process stores the vectors A_i^x into a two dimensional array Store[ ][ ] with $N \times N$ elements such that the *i*th row of the array Store[ ][ ] contains the vector 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] = $\sum$ Store[k][i], $(0 \leq k \leq N-1)$. Note that this updated Sum[i] denotes the total number of messages received by all other processes so far from process P_i.

Example 1: Consider the system shown in Fig. 2. Examine the diagram (left of the dotted line). At the starting states of the processes P_0 and P_1, the entries in their respective

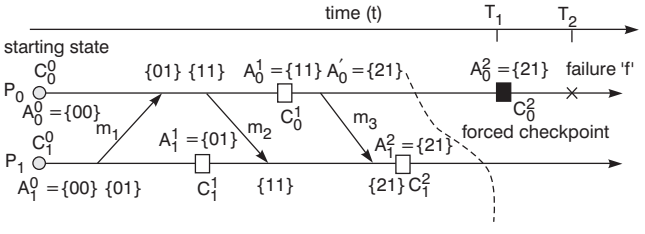

Fig. 2 *Distributed system with two processes*

Fig. 3 *Consistency check for C_0^1 and C_1^2*

vectors are all initialised to zero. After receiving the message m_1 from P_1, process P_0 updates its vector, i.e. A_0 becomes {01}. When P_0 sends the message m_2, it again updates A_0 to {11}. When process P_0 takes the checkpoint C_0^1, it also stores the vector A_0^1 (= {11}) together with the checkpoint in its disk. In this diagram, C_0^1 is the latest basic checkpoint taken by process P_0, after which it has sent one more message, m_3 to P_1. It thus updates vector A_0 to {21}, which we denote by A_0'. In a similar way, creation of the vectors by P_1 can be explained.

We now explain the formation of the vector sum and the interpretation of its different entries using this example. Consider the latest basic checkpoints of the processes. These are C_0^1 and C_1^2 with the respective vectors being A_0^1 and A_1^2. The array Store[ ][ ] is updated such that its 0th row contains the vector A_0^1 and 1st row contains the vector A_1^2. Then the elements Store [0][0] and Store [1][1] are set to zero. Finally the vector sum is updated using the relation Sum[i] = $\sum$ Store[k][i], $(0 \leq k \leq N-1)$. This is shown in Fig. 3.

We observe the following from Fig. 3:

(i) $A_0^1[0] = 1 =$ Sum[0] $- 1 = 2 - 1$; and
(ii) $A_1^2[1] = 1 =$ Sum[1];

From (i), it is clear that process P_1 has so far received two messages, namely, m_2 and m_3 from process P_0 (since Sum[0] = 2). However, P_0 has recorded in C_0^1 the sending of only message m_2. Therefore, the message m_3 sent by P_0 still remains an orphan until P_0 takes another checkpoint. From (ii), we can see that process P_0 has received only one message (m_1) from process P_1 and it has also been recorded by process P_1 at C_1^1. So, the message m_1 is not an orphan message.

4 Necessary observations

The following observations will help in designing the algorithm to find the globally consistent checkpoints (GCCs).

Observation 1: Given the set of the latest checkpoints, $S = \{C_i^m\}$ $(0 \leq i \leq N-1)$ and the vector sum, if $A_i^m[i] =$ Sum[i] $- d$ $(d > 0$, i.e. $A_i^m[i] <$ Sum [i]), then process P_i has sent d orphan messages after its latest checkpoint $C_i^m \in S$.

Observation 2: Given the set of the latest checkpoints, $S = \{C_i^m\}$ $(0 \leq i \leq N-1)$ and the vector sum, if $A_i^m[i] =$ Sum[i] $+ d$ $(d \geq 0$, i.e. $A_i^m[i] \geq$ Sum[i]), then these d messages sent by process P_i after its latest checkpoint C_i^m cannot be orphan messages.

Observation 3: Given the set of the latest checkpoints, $S = \{C_i^m\}$ $(0 \leq i \leq N-1)$ and the vector sum, if $A_i^m[i] \geq$ Sum[i] for all i $(0 \leq i \leq N-1)$, then the set S consists of N globally consistent checkpoints.

5 Significance of forced checkpoints

The concept of a forced checkpoint [12, 13] has been used in this work to design the algorithm. Consider the system shown in Fig. 2 (ignore the existence of the checkpoint C_0^2

for the time being). Suppose at time T_2 a failure 'f' occurs. If the recovery idea inherent in the asynchronous approach is followed, the processes P_0 and P_1 will restart their computation from C_0^1 and C_1^1, since these two checkpoints form the latest set of the GCCs (note that message m_2 will be treated as a lost message). Note that in such a recovery approach processes may have to roll back even to their respective starting states in the worst case. Now consider a different approach. Suppose, at time T_1 ($T_1 < T_2$), an attempt is made to determine the GCCs using the idea of forced checkpoints [12]. We start with the checkpoints C_0^1 and C_1^2, and find that the message m_3 is an orphan, because $A_0^1[0] = 1 = \text{Sum}[0] - 1$ (from Section 3). Thus C_0^1 and C_1^2 cannot be consistent. The consistent checkpoints are C_0^1 and C_1^1. 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), then this newly created checkpoint C_0^2 with its updated vector $A_0^2 = \{21\}$ becomes consistent with C_1^2, since $A_0^2[0] = \text{Sum}[0] = 2$. This means that message m_3 can no longer be an orphan message. Now if a failure 'f' occurs at time T_2, then after recovery processes P_0 and P_1 can simply roll back to their respective consistent states C_0^2 and C_1^2. Note 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. This also means that these two latest checkpoints form a recent consistent global checkpoint as in the synchronous approach. Therefore, use of the concept of forced checkpoints guarantees that after the system recovers from a failure the restarting state will always be the recent consistent global checkpoint and that there will be no domino effect. This makes the recovery process very simple. In other words, it can be said that by using forced checkpoints the idea of roll-forward [14] is implemented implicitly to reduce the amount of rollback to an appreciable extent whenever possible. In this context, it is worth mentioning that taking a forced checkpoint is no different from taking any other kind of checkpoint, i.e. a process just needs to save its state in a checkpoint.

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 corresponding to its previous checkpoint C_i^m belonging to the set $A_i^m[i] < \text{Sum}[i]$.

From the above discussion, it becomes clear that if we follow the idea of determining the set of the GCCs periodically by incorporating the concept of forced checkpoints then substantial reduction in the amount of rollback may be achieved. It also ensures that the effect of the domino phenomenon will be limited by the chosen time interval between successive invocations of the algorithm to find a consistent global state. Moreover, the recovery process becomes as simple as that in the synchronous approach even though the processes need not be synchronised while the GCCs are determined. The above discussion leads to the following observation stated below in Proposition 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 checkpoint. If this is the case, a message sent by P_i between these two checkpoints can never be an orphan message, although it may be a lost message.

We shall now prove that the set of the latest N checkpoints (including both the forced ones as well as the basic ones of those processes that do not need to take forced checkpoints) is the set of the globally consistent checkpoints. Let the set be S*. We create another set $S^A = \{A_i^m\}$ ($0 \le i \le N-1$), where the N vectors in S^A correspond to the N checkpoints in set S*.

Theorem 1: The N checkpoints in S* are globally consistent.

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

Case 1: A_i^m corresponds to a forced checkpoint of process P_i:

Since the checkpoint is a forced one, $A_i^m[i] \ge \text{Sum}[i]$. Hence, according to proposition 1, none of the messages sent by process P_i can remain an orphan.

Case 2: A_i^m corresponds to a latest basic checkpoint of process P_i:

Since the corresponding checkpoint is not a forced one, i.e. process P_i does not need to take a forced checkpoint, $A_i^m[i] \ge \text{Sum}[i]$, i.e. there is no orphan message from process P_i to any other process (observation 2).

These two cases confirm the condition $A_i^m[i] \ge \text{Sum}[i]$ ($0 \le i \le N-1$). Thus, according to observation 3, the N checkpoints in S* are globally consistent checkpoints.

6 Blocking approach

The algorithm to determine the globally consistent checkpoints is executed periodically by an initiator process P (different from the application processes) on any one of the processors in the system. It creates the GCCs beforehand so that whenever a failure occurs the system already knows where to roll back after recovery from the failure. The following assumptions are made:

1. The application program is suspended, i.e. blocked, when the algorithm is executed.
2. Processor failure may occur during the execution of the algorithm. Under the assumption given in Section 3, news of the failure reaches all other processors in finite time and the execution of the algorithm is aborted immediately. After recovery, the processes roll back to their respective globally consistent checkpoints created and saved during the previous execution of the algorithm.

The initiator process maintains a counter initialised with the value I. The counter is decreased at each time step. When the counter becomes zero, the initiator process starts executing the algorithm. Thus the value I is the time period between successive invocations of the algorithm. This period I is much greater than the time period t_{bi} required by any process P_i to take its basic checkpoints, i.e. $I \gg t_{bi}$, $0 \le i \le N-1$. This assumption about the value of I is reasonable for reliable systems. After the GCCs are determined, the counter is reset again to its initial value I.

We shall use the following notations in this algorithm. Let C_i^x be the latest checkpoint (basic) of process P_i, and A_i^x be the vector stored with the checkpoint. Note that A_i' and A_i^x may be identical in the event that process P_i has either not sent or received any message after taking its latest checkpoint C_i^x.

Algorithm A: The responsibilities of the initiator process P and each participating process P_i are stated below.

Initiator process P: Step 1: The initiator process P requests each application process P_i to send the vectors A_i^x and A_i'.

It forms the sets S and S′, such that $S = \{A_i^x\}$, $0 \le i \le N-1$, where A_i^x is the vector at the latest checkpoint of process P_i and $S' = \{A_i'\}$, $0 \le i \le N-1$, where A_i' is the last vector created by P_i after its latest checkpoint.

Step 2: The initiator process P updates the array Store[ ][ ] using S as the *i*th row of Store[ ][ ] $= A_i^x$ and then sets Store[i][i] $= 0$ for all i. The vector sum is then updated as Sum[i] $= \sum$ Store[k][i], $(0 \le k \le N-1)$ for all i.

Step 3: Set Flag = FALSE;

Do
If $A_i^x[i] \ge$ Sum[i] for all i, then
Set Flag = TRUE and the initiator process P informs every process P_i whether to take a forced checkpoint C_i^{x+1} after P_i's latest checkpoint.
Counter = I.
Algorithm terminates;
Else if for a process P_i, $A_i^x[i] =$ Sum[i] $-$ d(d > 0) (i.e. condition C), then
The initiator process P records that such a process P_i needs to take a forced checkpoint (i.e. C_i^{x+1}) where A_i^{x+1} should be equal to A_i' ($\in S'$). The initiator process P creates a new sum after replacing every such A_i^x in the Store[ ][ ] array with A_i^{x+1}.
In the set S, any such A_i^x is replaced as new $A_i^x = A_i^{x+1} = A_i'$.
End if
Until Flag = TRUE

Process P_i:
If P asks P_i to take a forced checkpoint, then
P_i takes the forced checkpoint C_i^{x+1} and stores it in the stable storage together with the vector $A_i^{x+1} = A_i'$.
Else P_i copies its checkpoint C_i^x and the vector A_i^x from the disk unit into the stable storage.
End if

The algorithm repeats the execution of step 3 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 so. Note that if a process P_k has not sent any message after taking its latest checkpoint (basic) it will not take any forced checkpoint irrespective of what other processes are doing. It is clear that once a process P_i has taken a forced checkpoint then condition C becomes false for P_i, i.e. none of the messages sent by P_i can remain an orphan.

Note that at any given time only one checkpoint (GCC) per process remains stored in the stable storage. Thus the latest GCCs are always stored in the same portion of the stable storage after flushing the last GCCs. Also, since the algorithm considers only the latest checkpoints, there is no need to store two or more basic checkpoints taken by a process P_i during time t_{bi}. Therefore, during the interval between two consecutive invocations of the algorithm, any current checkpoint to be stored in the disk unit can replace the already existing one in the unit (as stated in Section 3.2). Therefore, at any given time, at most two checkpoints per process remain stored – one (a GCC) in the stable storage, another in the disk unit. For example, consider the 'Else' statement of Process P_i's responsibility. This statement means that the checkpoint C_i^x (already stored in the disk unit) is identified by the algorithm as a GCC. Therefore, it is copied into the stable storage from the disk unit (as stated in Section 3.2). Now the next basic checkpoint that will be taken by process P_i before the next invocation of the algorithm will be stored in the disk unit.

Proof of correctness: In step 3, the algorithm first tests the condition stated in observation 3 (i.e. $A_i^x[i] \ge$ Sum[i]) for all processes. If the condition is true for all processes, it guarantees that the set of its latest checkpoints is the set of the GCCs according to observation 3; otherwise the algorithm applies condition C (i.e. $A_i^x[i] =$ Sum[i] $-$ d (d > 0)) to find which process(es) need(s) to take a forced checkpoint. According to proposition 1, the creation of such a forced checkpoint guarantees that no message sent by the corresponding process can remain an orphan. Thus, in the worst case, the algorithm may need to repeat N times the execution of step 3 to ensure that condition C becomes false for all N processes. This also guarantees the termination of the algorithm, resulting in a set of checkpoints (including both the forced ones and the basic ones of those processes that do not need to take forced checkpoints), which eventually becomes the set of the GCCs according to theorem 1.

Recovery: As in the synchronous approach, after the system recovers from a failure all the processes restart their computation from their respective GCCs, saved in the stable storage during the last execution of the algorithm.

The following example shows the case when all but one process need to take forced checkpoints.

Example 2: Consider the distributed system shown in Fig. 4. Suppose that at time T the algorithm is invoked. To start with, the algorithm creates the vector Sum = {010}. In step 3, it finds that $A_1^0[1] =$ Sum[1] $-$ 1. This means that the message m_1 sent by process P_1 to process P_2 is an orphan. The initiator process records this information that P_1 needs to take a forced checkpoint. However, the algorithm needs to ensure that if P_1 takes a forced checkpoint none of the messages sent by P_0 to P_1 can remain orphans. To do so the algorithm does the following. It creates the new Store[ ][ ] replacing the vector A_1^0 ($=\{000\}$) by A_1^1 ($=\{110\}$). The new sum becomes {110}. The algorithm repeats step 3. It finds that $A_0^1[0] =$ Sum[0] $-$ 1, i.e. that the message m_2 sent by process P_0 to process P_1 is an orphan. Finally, the initiator process P asks processes P_0 and P_1 to take their respective forced checkpoints (i.e. C_0^2 and C_1^1). P_0 and P_1 act accordingly. P_0 stores its vector A_0^2 ($= A_0' = \{100\}$) together with the checkpoint C_0^2 in the stable storage. Process P_1 also stores its vector A_1^1 ($= A_1' = \{110\}$) together with its forced checkpoint C_1^1 in the stable storage. Thus, the message m_1 and m_2 can no longer remain orphans. The system does not therefore have any orphan message. Thus the set of the checkpoints, i.e. $\{C_0^2, C_1^1, C_2^2\}$, is the set of the GCCs. If a failure now occurs before the next invocation of the algorithm, after recovery the processes P_0, P_1, and P_2 restart their computation from C_0^2, C_1^1, and C_2^2, respectively. Thus recovery is very simple.

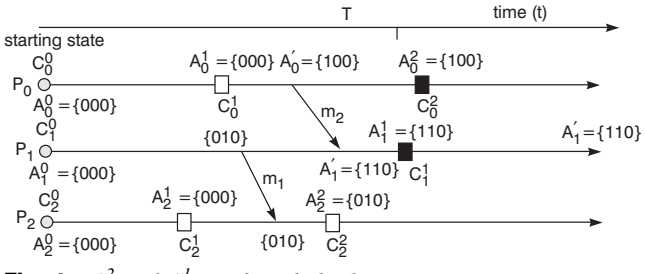

Fig. 4 *C_0^2 and C_1^1 are forced checkpoints*

It is easy to create an example that can show that in the worst case all processes are forced to take checkpoints. Thus, the main advantage that the presented approach offers compared to the synchronous approach is that, for an application with N processes, only in the worst case do all N processes need to take forced checkpoints, whereas in the synchronous approach N processes always need to take co-ordinated checkpoints.

When compared to the asynchronous and the communication-induced approaches, the presented blocking approach offers the following advantages:

(1) No piggybacking of any extra information.
(2) Effect of the domino phenomenon is limited by the time interval I between successive invocations of the algorithm.
(3) Recovery is as simple as in the synchronous approach.

7 Non-blocking approach

In this approach, the application program is not suspended while the initiator process executes the algorithm to determine a consistent global checkpoint. The algorithm is thus non-intrusive [15]. The responsibility of the initiator process remains the same. However, the responsibility of every process P_i is different here from that in the blocking approach. To explain the idea clearly, let us assume that the initiator process P starts and finishes its execution at times t_1 and t_2, respectively.

Since the computations performed by the different processes continue while the algorithm is executed, the following scenario is possible. During the interval $(t_2 - t_1)$, suppose P_j has sent a message m to P_i. This may lead to inconsistency in the global checkpoint if at time t_2 the initiator process P asks P_i to take a forced checkpoint and asks P_j not to take a forced checkpoint. The reason behind this possible inconsistency is that the message m may become an orphan depending on whether P_i has received it or not during the interval $(t_2 - t_1)$ and whether P_j has taken a basic checkpoint in the interval after sending the message m. This problem can be solved by using the following idea, which is a modification of the non-blocking idea reported in [15]. The following data structures are needed.

Every process includes a Boolean variable, say B (set at 1), in only the first computation (application) message it sends to every other process during the interval $(t_2 - t_1)$, i.e. after t_1 and before this process hears anything from the initiator process. Each process maintains separately a Boolean array of size N (for an N-process system) recording whether a message has previously been sent by it to each of the other processes in the current interval $(t_2 - t_1)$. All entries of all such arrays are initialised with 0s. In the Boolean array of a process P_j the *i*th entry will be set equal to 1 when P_j sends its first computation message to another process P_i during the interval. Any process P_j, prior to sending a computation message to any other process P_i during the interval $(t_2 - t_1)$, checks whether the *i*th entry in its array is set at 0. If it is, then it resets the entry at 1 (this means that this computation message is the first such message to be sent by P_j to P_i in the interval). Note that this is an extra overhead in the non-blocking approach compared to the blocking approach.

Let us now consider the different possible cases as stated below.

Case 1: Suppose that during the interval $(t_2 - t_1)$ a process P_i receives a message m with $B = 1$ and it is the first such message that P_i has received in the interval. Also, assume that P_i has not taken any basic checkpoint in the interval $(t_2 - t_1)$ before receiving this message. P_i then takes a new checkpoint C_i^{x+1} in the disk before processing the message. It updates the vector A_i' and stores it as A_i^{x+1} at the checkpoint C_i^{x+1}. Process P_i does not take any further checkpoint during the time $(t_2 - t_1)$ because otherwise the message m will become an orphan. It ensures that no such message (such as message m) received by P_i during the interval can remain an orphan. However, P_i goes on updating its vector when it receives or sends any message during the interval.

Case 2: If during the interval $(t_2 - t_1)$, P_i has already taken a basic checkpoint C_i^{x+1} before receiving any message with $B = 1$, it does not take any further checkpoint during the interval. As in case 1, it also ensures that no message m with $B = 1$ received by P_i during the interval can remain an orphan. As in the first case, process P_i updates its vector when it receives or sends any message during the interval.

Later at time t_2, if the initiator process P asks P_i to take a forced checkpoint, then P_i considers its checkpoint C_i^{x+1}, which it has already taken in the disk unit during the interval $(t_2 - t_1)$, as the 'already taken' forced checkpoint (which eventually becomes a GCC) and stores it in the stable storage at time t_2. On the other hand, suppose that process P decides that P_i does not need to take a forced checkpoint. Then irrespective of whether P_i has already taken a checkpoint during the interval $(t_2 - t_1)$ as in cases 1 and 2, P_i considers its previous checkpoint C_i^x to be a GCC (i.e. the last checkpoint P_i has taken in the disk unit before t_1). P_i then copies this GCC from the disk into the stable storage at time t_2. Note that a process P_i effectively may take at most one checkpoint during the interval. Also note that if process P_i receives a message with $B = 1$ after the GCCs are determined (i.e. after the time t_2) then it just processes the message.

The tradeoff between the blocking and the non-blocking approaches are: (i) in the latter approach there is the possibility of taking extra checkpoints in the interval $(t_2 - t_1)$, despite the fact that some or all these extra checkpoints may not belong to the set of the GCCs as determined by the algorithm. (ii) In the non-blocking approach, if process P_i takes a checkpoint C_i^{x+1} in the interval $(t_2 - t_1)$, say at time t*, and if this checkpoint happens to be a GCC, then the amount of the excess rollback that P_i suffers compared to that in the blocking approach is $(t_2 - t^*)$. The reason is that, in the blocking approach, P_i takes C_i^{x+1} (a forced one) at time t_2 whereas in the latter approach P_i takes C_i^{x+1} at time $t^* < t_2$. (iii) In the non-blocking approach, during the execution of the algorithm, a maximum of three checkpoints per process need to be stored. These are as follows: the first is its last GCC and this checkpoint is stored in the stable storage; the second is used by the initiator process to determine the GCCs and is stored in the disk unit of the processor running the process; the third, which can be taken during the interval $(t_2 - t_1)$, is stored in the disk unit in the same way as the second. In the blocking approach, a maximum of two checkpoints need to be stored, as mentioned earlier. (iv) In the non-blocking approach, the addition of a Boolean variable to the application messages is necessary for the correct behaviour of the algorithm, whereas in the blocking approach no piggybacking of any extra information with the application messages is necessary. (v) In the non-blocking approach, each process has to maintain separately a Boolean array of size N for an N-process system. This extra overhead is

absent in the blocking approach. (vi) The main advantage of the non-blocking approach is that processes are not suspended during execution of the algorithm. This is particularly important because the algorithm runs periodically and during each run the processes are never blocked. This definitely helps in achieving faster completion of the application program compared to the blocking approach.

8 Performance

The performance of the algorithm, considering its message complexity and computational complexity, is stated below.

Message complexity: In a distributed system with N application processes, the initiator process P asks all application processes to send their vectors and the application processes in turn do so. Later the initiator process sends a reply to each application process, informing each whether it needs to take a forced checkpoint. There are thus 3N messages that need to be exchanged between the initiator process and the application processes. As stated earlier in Section 3.2, in this work we have assumed that a process P_i copies its checkpoint C_i^x together with the corresponding vector from the disk unit into the stable storage of the distributed system if the checkpoint C_i^x belongs to the set of the GCCs as determined by the algorithm. Therefore, in both the blocking and non-blocking approaches, irrespective of whether a process needs to take a forced checkpoint, every process has to store its latest GCC as determined by the algorithm in the stable storage. Thus there are N more messages that are needed for storing the N number of GCCs. Therefore, altogether 4N messages are needed for the successful termination of the algorithm. Hence, the message complexity is $O(N)$.

Computational complexity: Creating the sum once (to check if a process needs to take a forced checkpoint) will have $O(N^2)$ complexity. In the worst case, all N processes need to take forced checkpoints. Therefore, the complexity in the worst case is $O(N^3)$.

Note that the frequency of running the algorithm (as determined by the value I of the counter) also has some effect on the performance. For example, if the system is highly reliable then the algorithm does not need to run frequently (i.e. higher value of I); in this case, the system will execute the application program quicker. On the other hand, if the system is not highly reliable, then frequent execution of the algorithm (i.e. I is small) may be required, which in effect will slow down execution of the application program.

9 Elimination of useless checkpoints

In this work, the presented algorithms to determine a consistent global checkpoint need only consider the last (latest) basic checkpoints taken by the processes in the interval I. All other checkpoints taken in the interval prior to these last checkpoints can never be the GCCs. However, some of the latest checkpoints may not be the GCCs, despite the fact that without them the algorithms cannot determine a consistent global checkpoint. Viewed is this light, none of the latest checkpoints taken in the interval I can be termed as really useless. In general, any checkpoint that cannot be a GCC is a useless checkpoint [2, 9]. A simple solution to avoid taking useless checkpoints is stated below.

Suppose that at the start of the application program each participating process P_i is informed about the value of the period I and is asked to take only one checkpoint in each interval I and always at time (I-t) from the previous termination of the algorithm (where 't' is a small value). Then the algorithm, whether blocking or non-blocking, will consider these as the latest checkpoints and use them to determine the GCCs. This saves time by not taking useless checkpoints. It may appear that all the processes need to take their respective checkpoints exactly at the same time. This is not true because the present work does not need the processes to be synchronised at all while the checkpoints are taken.

10 Conclusion

In this paper, a new roll-forward scheme for checkpointing/recovery has been presented. We have presented some important and interesting theoretical results. The two main important aspects are: the use of the direct-dependency concept for basic checkpoints and the use of the forced checkpoint concept. Our work has the following advantages: recovery is as simple as in the synchronous approach despite the fact that, unlike in the synchronous approach, processes need not be synchronised while the GCCs are determined, and the amount of rollback is limited by the value of the counter. Also note that although the direct-dependency concept is used, in the blocking approach there is still no piggybacking of any extra information with the application messages. Both the blocking and non-blocking approaches have been presented and their relative advantages have been discussed. We have also shown that the creation of useless checkpoints can be avoided in a simple way.

11 Acknowledgment

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

12 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 BALDONI, R., QUAGLIA, F., and FORNARA, P.: 'An index-based checkpointing algorithm for autonomous distributed systems', *IEEE Trans. Parallel Distrib. Syst.*, 1999, **10**, (2), pp. 181–192
3 SINGHAL, M., and SHIVARATRI, N.G.: 'Advanced concepts in operating systems' (McGraw-Hill, 1994)
4 KOO, R., and TOUEG, S.: 'Checkpointing and rollback-recovery for distributed systems', *IEEE Trans. Softw. Eng.*, 1987, **SE-13**, (1), pp. 23–31
5 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
6 GARG, V.K.: 'Principles of distributed systems' (Kluwer Academic Publishers, 1996)
7 TSAI, J., KUO, S.-Y., and WANG, Y.-M.: 'Theoretical analysis for communication-induced checkpointing protocols with rollback-dependency trackability', *IEEE Trans. Parallel Distrib. Syst.*, 1998, **9**, (10), pp. 963–971
8 BALDONI, R., HELWAY, J.M., MOSTERFAOUI, A., and RAYNAL, M.: 'A communication-induced checkpointing protocol that ensures rollback dependency trackability'. Proceedings of IEEE international symposium on Fault-tolerant computing, 1997, pp. 68–77
9 HELARY, J.M., MOSTERFAOUI, A., NETZER, R.H.B., and RAYNAL, M.: 'Communication-based prevention of useless checkpoints in distributed computations', *Distrib. Comput.*, 2000, **13**, (1), pp. 29–43
10 CAO, G., and SINGHAL, M.: 'On coordinated checkpointing in distributed systems', *IEEE Trans. Parallel Distrib. Syst.*, 1998, **9**, (12), pp. 1213–1225

11 MANIVANNAN, D., and SINGHAL, M.: 'Quasi-synchronous checkpointing: models, characterization, and classification', *IEEE Trans. Parallel Distrib. Syst.*, 1999, **10**, (7), pp. 703–713
12 ZENG, L., and GUPTA, B.: 'Efficient utilization of bandwidth and storage capacity for recovery in mobile computing environment'. Proceedings of symposium on Performance evaluation of computer and telecommunication systems, Chicago, USA, July 1999, pp. 97–101
13 GASS, R.C., and GUPTA, B.: 'An efficient checkpointing scheme for mobile computing systems'. Proceedings of ISCA 13th international conference on Computer applications in industry and engineering, Honolulu, USA, November 2000, pp. 323–328
14 PRADHAN, D.K., and VAIDYA, N.H.: 'Roll-forward checkpointing scheme: a novel fault-tolerant architecture', *IEEE Trans. Comput.*, 1994, **43**, (10), pp. 1163–1174
15 PRAKASH, R., and SINGHAL, M.: 'Low-cost checkpointing and failure recovery in mobile computing systems', *IEEE Trans. Parallel Distrib. Syst.*, 1996, **7**, (10), pp. 1035–1048
16 GUPTA, B., BANERJEE, S.K., and LIU, B.: 'A pseudo-synchronous roll-forward recovery approach using basic checkpoints for distributed systems'. Proceedings of ISCA 16th international conference on Computers and their applications, Seattle, USA, March 2001, pp. 476–480