A Novel Roll-Back Mechanism for Performance Enhancement of Asynchronous Checkpointing and Recovery

Bidyut Gupta and Shahram Rahimi
Department of Computer Science, Southern Illinois University
Mail Code 4511, Carbondale, IL 62901-4511, USA
{bidyut, rahimi}@cs.siu.edu

Yixin Yang
Department of Biological Sciences, Emporia State University
Emporia, KS 66801, USA
yyang@emporia.edu

Keywords: asynchronous checkpointing, recovery, maximum consistent state

Received: May 26, 2006

In this paper, we present a high performance recovery algorithm for distributed systems in which checkpoints are taken asynchronously. It offers fast determination of the recent consistent global checkpoint (maximum consistent state) of a distributed system after the system recovers from a failure. The main feature of the proposed recovery algorithm is that it avoids to a good extent unnecessary comparisons of checkpoints while testing for their mutual consistency. The algorithm is executed simultaneously by all participating processes, which ensures its fast execution. Moreover, we have presented an enhancement of the proposed recovery idea to put a limit on the dynamically growing lengths of the data structures used. It further reduces the number of comparisons necessary to determine a recent consistent state and thereby reducing further the time of completion of the recovery algorithm. Finally, it is shown that the proposed algorithm offers better performance compared to some related existing works that use asynchronous checkpointing.

Povzetek: Opisan je izboljšan postopek okrevanja v porazdeljenih sistemih.

1 Introduction

Checkpointing and rollback-recovery are well-known techniques for providing fault-tolerance in distributed systems [1]-[5]. The failures are basically transient in nature such as hardware error [1]. Typically, in distributed systems, all the sites save their local states, known as local checkpoints. All the local checkpoints, one from each site, collectively form a global checkpoint. A global checkpoint is consistent if no message is sent after a checkpoint of the set and received before another checkpoint of the set [2]-[4], that is, each message recorded as received in a checkpoint should also be recorded as sent in another checkpoint. In this context, it may be mentioned that a message is called an orphan message if it is recorded as received in a checkpoint, but not recorded as sent in another checkpoint. The local checkpoints belonging to a consistent global checkpoint will be termed in the present work as globally consistent checkpoints (GCCs). After recovery from a failure processes in a distributed computation restart their computation from a consistent global checkpoint /state (CGS) of the system, i.e. from their respective GCCs. It may be noted that a consistent global checkpoint of a system is termed as a recent or a maximum one if, after the system recovers from a failure, the number of events (states) rolled back at each processor is a minimum [6].

To determine consistent global checkpoints, two fundamental approaches have been reported in the literature [1]-[9]. These are synchronous and asynchronous approaches. In the synchronous approach, processes involved coordinate their local checkpoint actions such that the set of all recent checkpoints in the system is guaranteed to be consistent. Although it simplifies recovery it has the following disadvantages: (1) additional messages need to be exchanged by the checkpointing algorithm when it takes each checkpoint; (2) synchronization delay is introduced during normal operation [5]. In the asynchronous approach, processes take checkpoints independently without any synchronization among them. Therefore, it is the simplest form of taking checkpoints. However, because of the absence of synchronization there is no guarantee that a set of local checkpoints taken will be a consistent set of checkpoints. That is, there may exist orphan messages between the local checkpoints. In order to get rid of the orphan messages while determining the GCCs, processes have to rollback. In such a situation, rolling back one process causes one or more other processes to roll back. This effect is known as the domino effect [5]. This is the main drawback of the asynchronous approach. So, a recovery algorithm has to search for the most recent consistent set of checkpoints before the system restarts its normal operation. Therefore, the recovery process is

quite complex while the checkpointing scheme is much simpler compared to the same in synchronous approach.

2 Related Works

In this work, we have considered asynchronous checkpointing approach because of its simplicity in taking checkpoints. So, in this section we state briefly the contributions of some noted related works. When processes take checkpoints independently, some or all of the checkpoints taken may be useless for the purpose of constructing consistent global checkpoints. A set of checkpoints can belong to the same consistent global snapshot if no zigzag path (Z-path) exists from a checkpoint to any other checkpoint [15]. In other words, absence of a Z-path means absence of any orphan message. A theoretical framework for characterizing quasi-synchronous algorithms has been presented in [12]. Quasi-synchronous checkpointing algorithms reduce the number of useless checkpoints by preventing the formation of noncausal Z-paths between checkpoints and advance recovery line. "Advancement of recovery line" is interpreted as follows: the more the recovery line is advanced, the less is the amount of computation to be redone by processes after the system of processes restart their normal operation; meaning thereby the reduction in the amount of rollback per process after the system recovers from failure. Depending on the degree to which the non causal Z-paths are prevented, quasi-synchronous checkpointing algorithms are classified into three classes namely [12], Strictly Z-Path Free (SZPF), Z-Path Free (ZPF), and Z-Cycle Free (ZCF).

Manivannan and Singhal [13] have presented a quasi-synchronous checkpointing algorithm which allows the processes to take checkpoints asynchronously and reduces the number of useless checkpoints by forcing processes to take additional checkpoints. In this checkpointing algorithm, each process maintains a counter which is periodically incremented and the time period is same in all the processes. When a process takes a checkpoint, it assigns the current value of its counter as the sequence number for the checkpoint. Each message is equipped (i.e. piggybacked) with the sequence number of the current checkpoint. If the sequence number accompanying the message is greater than the sequence number of the current checkpoint of the process receiving the message, then the receiving process takes a checkpoint and assigns the sequence number received in the message as the sequence number to the new checkpoint and then processes the message. Quasi-synchronous checkpointing algorithm makes sure that none of the checkpoints taken lies on a Z-cycle in order to make all checkpoints useful. Asynchronous recovery algorithms are also presented in this paper based on the checkpointing algorithm. A failed process needs to roll back to its latest checkpoint and requests other processes to rollback to their consistent (latest) checkpoints. The work claims to be free from any domino effect. However, arguably this work is more of a synchronous approach than an asynchronous approach; partly because all processes have identical time periods to take checkpoints, and checkpoint sequence numbers are used so that all the i^{th} checkpoints of all processes are taken at the same time (i.e., logically at same time). Hence, we argue that there is no question of domino effect as this work is not at all an asynchronous approach.

Gupta et al. [11] have proposed a hybrid roll forward checkpointing/recovery approach. Processes take checkpoints periodically and these time periods are different for different processes. Periodically, in absence of any failure, an initiator process invokes the algorithm to advance the recovery line; the duration of this period is assumed to be much larger than the time period of any individual process. Therefore, the domino effect is limited by this time period. The main advantages of this work are that each process may need to keep at most two checkpoints at any time, processes participate in the algorithm simultaneously ensuring re-execution time after a failure is limited by the period of execution of the algorithm, and finally, recovery is as simple as in the synchronous checkpointing/recovery approach.

Ohara et al. [14] proposed an uncoordinated checkpointing algorithm for finding a recovery line where a given checkpoint is the earliest. In this algorithm, each process maintains a set of all local checkpoints on that process in a local vector. All local checkpoints which are just behind a given checkpoint are initially assumed to form a consistent global checkpoint. The algorithm checks happened-before relation for any coupled local checkpoints belonging to an ordered global checkpoint set. If there exists any happened-before relation, it replaces a local checkpoint with a successive local checkpoint of the same process. The algorithm may end by either finding a recovery line or running out of local checkpoints to be replaced.

Venkatesan and Juang [16] presented an asynchronous checkpointing algorithm where each process take checkpoints independently and keeps track of the number of messages it has sent to other processes as well as the number of messages it has received from other processes. The algorithm is initiated by the process which fails and is recovered from thereafter or when it learns about process failure. During its each iteration, a process needs to compare the number of messages received by it and the actual number of messages sent by the other process, at each of its checkpoint starting from the most recent one. The received vectors corresponding to all the checkpoints including the current one and the one where next iteration starts, need to be fetched from the storage in order to decide the checkpoint for the next iteration to start with.

3 System Model

The distributed system has the following characteristics [1], [6], [10]:

1. Processes do not share memory and they communicate via messages sent through channels.
2. Channels are made virtually lossless and order of the messages is preserved by some end-to-end transmission protocol.

3. When a process fails, all other processes are notified of the failure in finite time. We also assume that no further processor (process) failures occur during the execution of the algorithm. In fact, the algorithm must be restarted if there are further failures.

Below we state the problem considered in this work.

Problem Formulation: In this work, we have considered asynchronous checkpointing approach because of its simplicity in taking checkpoints. That is, processes take checkpoints periodically and each process determines independently its time period of taking its checkpoints. So, different processes may have different time periods for taking their checkpoints. After the system recovers from a failure, processes start from the recent consistent state of the system. However, the main drawback of this approach is that determining a consistent global checkpoint may involve a very large number of pairwise comparisons of checkpoints belonging to different processes because of the presence of a possible domino effect. In absence of any hybrid approach [11], in the worst case, all checkpoints of all processes may have to be compared. However, asynchronous checkpointing approach is suitable for highly reliable systems where failures occur very seldom.

In this work, our objective is to design an efficient recovery algorithm that will reduce considerably the number of unnecessary pairwise comparisons of checkpoints while determining a consistent global checkpoint. In other words, our objective is to identify a priori the checkpoints that can not be the GCCs so that we can exclude these checkpoints from comparison resulting in a fast determination of a recent consistent global checkpoint (state) of the system. Note that an initial version of this work has appeared in [17].

4 Data Structures

Let us assume that the distributed system under consideration consists of n processes. Each process P_i maintains a vectors V_i of length n. The V_i vector records the number of messages process P_i has sent to every other process with the exception that the element $v_{i,i}$ (=V_i (i)), i.e. the number of messages process P_i has sent to itself will be always zero. The V_i vector is described below:

$$V_i = [v_{i,0}, v_{i,1}, \ldots . , v_{i,i}, \ldots . , v_{i,n-1}]$$

where $v_{i,j} = V_i$ (j) and represents the number of messages sent by process P_i to process P_j, and $v_{i,i}$ is always zero.

All entries in V_i are initialized to zero. Each time process P_i decides to send a message m to process P_j, then V_i(j) is incremented by one. This facilitates process P_i to know how many messages it has sent to process P_j. In this work, $C_{j,r}$ represents the r^{th} checkpoint taken by process P_j. Sometimes when mentioning the checkpoint number is irrelevant, we simply use C_j to denote a checkpoint taken by P_j. Each process P_i also maintains a linear list R_i of dynamically growing length. At any given time t, the length of the list R_i (i.e. the number of the entries in the list) is equal to the number of checkpoints taken by P_i till time t. For example, the length of the list is 3 at the 3^{rd} checkpoint of process P_i where as its length will be 4 at its 4^{th} checkpoint and so on. The list R_i is described as $R_i = [r_{i,1}, \ldots . , r_{i,r}, \ldots\ldots]$, where $r_{i,r} = R_i$ (r) and represents the number of messages received by process P_i from all other processes till its r^{th} checkpoint. Each such list is initially empty.

Each process stores its vectors and the lists together with the corresponding checkpoints in stable storage. Also copies of the lists and the vectors are stored in the respective local memories of the processors running the processes. It offers their faster access than to access them from stable storage whenever possible. In addition, each process maintains a Boolean flag. This flag is used to convey some specific information (described later).

5 Observations

Consider the system of three processes P_1, P_2, and P_3 as shown in Fig. 1. The vectors V_1, V_2, and V_3 initially have all their entries set to zero. The lists R_1, R_2, and R_3 are initially all empty. By the time process P_1 takes its first checkpoint $C_{1,1}$, it did not send any message to P_2 or P_3. So its V_1 vector is [000]. Also, process P_1 received one message before it took its first checkpoint; so now the list R_1 has one entry, i.e. $R_1 = [1]$. By the time process P_1 takes its second checkpoint $C_{1,2}$, it has already sent one message to P_2. So it increments $V_1(2)$ by 1 and the vector V_1 is now = [010]. Also, process P_1 has not received any messages (from P_2 or from P_3) before it takes its second checkpoint. So the list R_1 at $C_{1,2}$ is [1,1]. In the same way, the vector and the list are updated at each checkpoint of each process. This example will be used later in this paper to illustrate the working principle of our proposed algorithm.

We assume that a process P_i after recovery from its failure acts as the initiator process, i.e., P_i is responsible for invoking the recovery algorithm. To start with P_i sends a message requesting all P_j, $0 \le j \le n-1$, $j \ne i$, to send to it their respective V_j vectors corresponding to their latest checkpoints. Upon receiving the request, every process P_j sends its V_j to P_i. After receiving the vector V_j from all processes the initiator process P_i forms a two dimensional array V_N. It is written below.

$$V_N = \begin{vmatrix} 0 & v_{0,1} & \cdots & v_{0,n-1} \\ v_{1,0} & 0 & \cdots & v_{1,n-1} \\ . & . & \cdots & . \\ v_{j,0} & v_{j,1} & \cdots & v_{j,n-1} \\ . & . & \cdots & . \\ v_{n-1,0} & v_{n-1,1} & \cdots & 0 \end{vmatrix}$$

where the j^{th} row represents V_j, $0 \le j \le n-1$. The initiator process then computes the column sums to create the following vector:

$$V_C = [v_c^0, v_c^1, \ldots, v_c^j, \ldots, v_c^{n-1}]$$

where v_c^j = column sum of the entries of the j^{th} column of V_N and is given as

$$v_c^j = V_C(j) = \sum V_N(i, j), \text{ for } i = 1 \text{ to } n.$$

Therefore, v_c^j represents the total number of messages sent to process P_j by all other processes as recorded in each sending process' latest checkpoint. The initiator process P_i then unicasts v_c^j (= $V_C(j)$) to process P_j. After receiving v_c^j from P_i, each process P_j computes $D_j = R_j(r) - v_c^j$, assuming that the last checkpoint of process P_j is the r^{th} checkpoint ($C_{j,r}$). The difference D_j (if >0) gives the exact number of orphan messages received by a process P_j till its checkpoint $C_{j,r}$, from all other processes in the system. Initiator process P_i also does similar computation to determine the exact number of orphan messages (if any) it has received till its latest checkpoint $C_{i,r}$. Proof of this statement is given later.

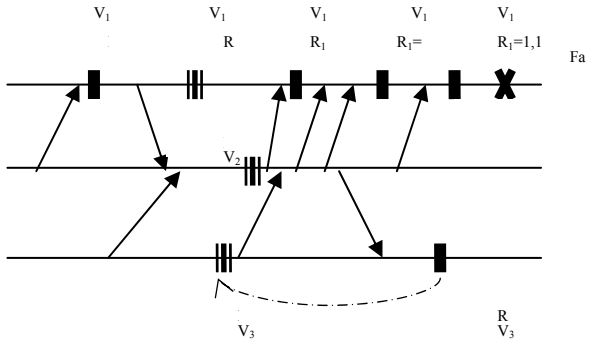

Figure 1: Vectors (V_i) and lists (R_i) for i = 1, 2, and 3

Observe that for every process P_j, v_c^j and $R_j(r)$ may not be identical, because some of the sent messages (recorded already by the sending processes at their respective latest checkpoints) may not have arrived yet at P_j (i.e. $v_c^j \geq R_j(r)$), or some of the received messages (by P_j) may not have been recorded at the latest checkpoints of some sending processes because these messages may have been sent after their latest checkpoints (i.e. $v_c^j \leq R_j(r)$).

Assume that the last checkpoint of process P_j is the r^{th} checkpoint ($C_{j,r}$) and D_j is greater than zero (D_j >0). Search in the list R_j is performed backwards, starting with its last component. Thus, we search the proceeding entries of the list R_j from $R_j(r)$ till the first $R_j(m)$ so that $R_j(r) - R_j(m) \geq D_j$, (m < r). Then, the checkpoints $C_{j,r}$, …, $C_{j,\,m+1}$ are excluded from the consideration of GCC composition, i.e. these checkpoints will be skipped. So, now we start from the checkpoint $C_{j,m}$ of process P_j. The vector V_j at checkpoint $C_{j,m}$ along with the Boolean flag "1" are sent to the initiator process P_i for the computation of the next iteration.

In the next iteration, if D_j is smaller than or equal to zero ($D_j \leq 0$), which means that process P_j has not received any orphan message till the checkpoint $C_{j,r}$. process P_j will send the flag "0" to the initiator process P_i. The initiator process P_i will use the vector V_j at $C_{j,r}$ for the computation of the next iteration. Initiator process P_i is also involved in similar computation like any other process P_j to determine its appropriate vector V_i needed for the computation of the next iteration. This will be repeated until all processes send "0" flags to the initiator process P_i and P_i's own flag is also 0 . Then the initiator process P_i will notify all processes to rollback to their respective latest checkpoints at which their corresponding flags have the value 0 each. Thus, this set of checkpoints is a globally consistent checkpoint (proof is given later).

The following observations are necessary for designing the recovery algorithm.

Lemma 1: Let $C_{j,r}$ be the latest checkpoint of process P_j at time t. If $D_j > 0$, then process P_j has received a total D_j number of orphan messages from other processes.

Proof: $R_j(r)$ represents the total number of messages process P_j has received so far from all other processes till time t. Also v_c^j represents the total number of messages sent by all other processes to P_j as recorded in their latest checkpoints. Therefore $D_j > 0$ means that at least some process P_i ($i \neq j$) has sent some message(s) to P_j after taking its latest checkpoints. It also means that the sending processes have not yet been able to record these D_j messages. Since all such D_j messages have been received and recorded in P_j's latest checkpoint, but remain unrecorded by the sending processes, therefore P_j has received D_j number of orphan messages from the rest of the processes with respect to the checkpoint $C_{j,r}$. ■

Lemma 2: If $D_j \leq 0$, process P_j has not received any orphan message.

Proof: $D_j = 0$ means that the number of messages received by P_j is equal to the number of messages sent to P_j and these sent (also received) messages have already been recorded by the sending processes in their latest checkpoints. Therefore the received messages can not be orphan.

Also, $D_j < 0$ means that the number of the messages received by P_j is less than the number of messages sent to it. Now v_c^j is the total number of messages sent by all other processes to P_j as recorded in the latest checkpoints of the sending processes. It means that all messages received by P_j have already been recorded by the senders. Hence none of such received messages can be an orphan. Hence the proof follows. ■

Lemma 3: Let $D_j > 0$ at the checkpoint $C_{j,r}$ of process P_j and let m denote the largest integer that satisfies $R_j(r) - R_j(m) \geq D_j$ (m < r). Then none of the checkpoints $C_{j,r}$, $C_{j,r-1}$, …, $C_{j,\,m+1}$ belongs to the set of the globally consistent checkpoints.

Proof: Because m is the largest integer that satisfies $R_j(r) - R_j(m) \geq D_j$ (m < r), the relation $R_j(r) - R_j(i) < D_j$ is established for any i ($m+1 \leq i \leq r$). Moreover, according to Lemma 1, P_j has received exactly D_j number of orphan messages from all other processes. So there must be at least one orphan message received by process P_j before $C_{j,r}$, and the same also is true before every checkpoint between $C_{j,r}$ and $C_{j,m}$. Hence, none of the checkpoints $C_{j,r}$, $C_{j,r-1}$, …, $C_{j,\,m+1}$ can belong to the set of the globally consistent checkpoints. ■

Theorem 1: Given a set $S^* = \{C_{j,r}\}$ of n checkpoints, one from each P_j, $0 \le j \le n-1$, if for every checkpoint $C_{j,r}$, its corresponding $D_j \le 0$, then S^* is the set of the globally consistent checkpoints.

Proof: Since $D_j \le 0$, for each process P_j, ($0 \le j \le n-1$) at its checkpoint $C_{j,r}$ ε S^*, therefore, all received messages by any such process P_j have already been recorded as sent by the sending processes in their corresponding checkpoints. Hence, according to Lemma 2 none of the messages received by process P_j is an orphan message. This is true for all processes. Therefore, the system of n processes does not have any orphan messages with respect to the checkpoints of the set S^*. Hence the set S^* is the set of globally consistent checkpoints. ■

Before we present the algorithm formally, we give an illustration of its working principle using the example of Fig. 1.

An illustration: Suppose a failure 'f' occurs on the processor running the process P_1. The process P_1 that became faulty, acts as the initiator after recovery from failure. After the system recovers from the failure, to start with, initiator process P_1 broadcasts a request asking the other two processes P_2 and P_3 to send their respective vectors V_2 and V_3 corresponding to their latest checkpoints $C_{2,1}$, and $C_{3,2}$. In this example, the three latest checkpoints of processes P_1, P_2, and P_3 before the failure occurs are $C_{1,5}$, $C_{2,1}$, and $C_{3,2}$. The respective vectors V_1, V_2, and V_3 at the three latest checkpoints are [010], [100] and [020]. After receiving all these vectors, P_1 (it becomes the initiator after recovery from failure) forms a two dimensional array V_N. It is written below:

$$V_N = \begin{vmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 2 & 0 \end{vmatrix}$$

P_1 creates the vector V_C = [130] and unicasts v_c^j to each process P_j, for j = 1, 2, and 3. After receiving v_c^j from P_i each process P_j computes D_j ($= R_j(r) - v_c^j$) (assuming the last checkpoint of P_j is the r^{th} checkpoint) to determine the total number of orphan messages (if any) it has received with respect to its latest checkpoint and also P_i does the same. The lists R_1, R_2, and R_3 at the latest checkpoints ($C_{1,5}$, $C_{2,1}$, and $C_{3,2}$) of processes P_1, P_2 and P_3 are [1,1,2,4,5], [2] and [0,1] respectively. P_1 finds that $D_1 = (5-1) = 4$; so it has received 4 orphan messages. It calculates the difference between $R_1(5)$ and $R_1(2)$ and finds that $R_1(5) - R_1(2) = 4 = D_1$; so process P_1 now considers the vector V_1 (= [010]) at $C_{1,2}$ along with a flag "1" for the computation of the next iteration. P_2 finds that it has not received any orphan message because $D_2 = (2-3) < 0$. So it sends the same vector [100] and a flag "0" to P_1. Process P_3 finds that $D_3 = (1-0) = 1$; so it has received an orphan message. It calculates the difference between $R_3(2)$ and $R_3(1)$ and finds that $R_3(2) - R_3(1) = 1 = D_3$; so process P_3 now sends the vector V_3 (= [010]) at $C_{3,1}$ along with a flag "1" to P_1 for the computation of the next iteration. In the second iteration, P_1 forms the following two dimensional array.

$$V_N = \begin{vmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{vmatrix}$$

P_1 creates the vector V_C = [120] and unicasts v_c^j to process P_j, for j = 1, 2, and 3. P_1 finds that it has not received any orphan message because at $C_{1,2}$, its $D_1 = 1 - 1 = 0$. So, it sets its flag to 0. P_2 also finds that it has not received any orphan message because at $C_{2,1}$, its $D_2 = 2 - 2 = 0$; and it sends the flag "0" to P_1. Similarly, P_3 finds that it has not received any orphan message because at $C_{3,1}$, its D_3 ($= R_3(1) - v_c^3$) $= 0 - 0 = 0$, and it sends a flag "0" to P_1. Thus, P_1 receives flag 0 from each process including its own flag set to 0. It then notifies each process to rollback to the current checkpoints corresponding to these flags (= 0). At this time, none of the processes needs to roll back further and hence P_1 terminates the algorithm. Thus the algorithm terminates after two iterations. Therefore the GCCs belonging to the maximum consistent state are $C_{1,2}$, $C_{2,1}$, and $C_{3,1}$.

It may be noted that in each iteration we need to fetch only the latest R_j for each process P_j and some V_j vectors (not all) to determine the GCCs. In each iteration, the checkpoints that can not be the GCCs are identified and their vectors V_j are not fetched at all. That is, the presented approach will not repeat its operation unnecessarily for these vectors corresponding to these non-GCCs. It definitely makes the approach fast and efficient. Observe what happens if we do not consider the above idea to determine the GCCs. It is stated below.

First, $C_{1,5}$, $C_{2,1}$, and $C_{3,2}$ are considered and compared pairwise to determine if they are globally consistent. Since $C_{1,5}$ and $C_{3,2}$ are not, so in the next iteration $C_{1,4}$, $C_{2,1}$, and $C_{3,1}$ are considered pairwise. But $C_{1,4}$ cannot be a GCC. Therefore $C_{1,3}$, $C_{2,1}$, and $C_{3,1}$ are now considered. But since $C_{1,3}$ can not be a GCC, therefore $C_{1,2}$, $C_{2,1}$, and $C_{3,1}$ are now considered. This time it is found that these three checkpoints are globally consistent. Therefore four iterations for pairwise comparisons of three checkpoints, one from each process, are needed to determine the GCCs as opposed to only two when the approach presented in this work is followed. It also means that the number of trips to the stable storage for fetching checkpoints can also be reduced to a good extent in the proposed approach. It definitely makes our algorithm fast. Moreover when processes take large number of checkpoints before a failure occurs, our approach may offer even much better performance from the viewpoint of a possible large reduction in the number of iterations (i.e. the number of trips to stable storage as well) to determine the GCCs. As a result, the recovery scheme also will be faster. Besides, it is clear from the example that each process P_j simultaneously identifies the checkpoints that cannot be globally consistent and therefore these checkpoints should be skipped. This parallelism of the algorithm further enhances the speed of execution of the recovery approach.

6 Algorithm to Determine Globally Consistent Checkpoints

In the following algorithm we assume that process P_i was faulty. So, it becomes the initiator of the recovery algorithm after it recovers from the failure.

6.1 Algorithm Recovery

Input: Given the latest n checkpoints, one for each process P_j, $0 \leq j \leq n-1$, for an n process system and the corresponding vectors V_j and lists R_j at these n checkpoints.

Output: A set of globally consistent checkpoints (maximum consistent state of the system).

The responsibilities of each participating process P_j and the initiator process P_i are stated in Fig. 2.

Proof of Correctness: Each process P_j repeats its steps 1, 2, 3, and 4 to arrive at a checkpoint that has not recorded the receipt of any orphan message from the other processes (using the observations of Lemmas 1, 2, and 3). In other words, it identifies the checkpoints that can not belong to the set of the globally consistent checkpoints and skips them. This decision is taken by identifying a checkpoint $C_{j,m}$ such that m is the largest integer that satisfies $R_j(r) - R_j(m) \geq D_j$ ($m < r$). None of the checkpoints $C_{j,r}$, $C_{j,r-1}$, …, $C_{j,m+1}$ can belong to the set of the globally consistent checkpoints and they are skipped. However, the initiator process P_i decides when to terminate the algorithm, i.e., when the checkpoints can become globally consistent. Process P_i checks to see if all processes send flags of 0, i.e. $D_j \leq 0$ for each process P_j. If so, the algorithm terminates according to Theorem 1. Note that the condition $D_j \leq 0$ must always occur during the execution of the algorithm. It may be observed that in the worst case, because of some typical communication pattern, the domino effect may force each process to restart from its initial state where for each process P_j we always have $D_j = 0$. Besides, since the algorithm starts with the latest checkpoints, the number of events (states) rolled back at each processor is a minimum. This is true because, in its Step 4 each process P_j skips only the checkpoints that are non GCCs. Thus the algorithm determines the maximum consistent state of the system as well. ■

6.2 Advantages of the proposed approach

The presented algorithm offers the following advantages. During its each iteration, each process P_j determines the checkpoints that can not be the GCCs. Therefore, the algorithm is able to avoid any unnecessary computations of V_C corresponding to these non GCCs. The presented algorithm skips checkpoints that do not belong to the set of the globally consistent checkpoints; thus it avoids many unnecessary pairwise comparisons. It also means that the number of trips to the stable storage for fetching checkpoints can also be reduced to a good extent in the proposed approach. It definitely makes the algorithm fast and efficient. The simultaneous execution of the algorithm by all participating processes also contributes to the speed of execution of the algorithm. Besides, the algorithm can find the maximum number of checkpoints to be skipped by determining the largest integer m, which satisfies $R_j(r) - R_j(m) \geq D_j$. This guarantees significant reduction in the iterations of computation.

6.3 Performance

Message complexity**:** Suppose the termination of the algorithm requires the construction of the vector V_C by the initiator process P_i to occur k times (i.e. k number of iterations). During each such time every process in the n-process system exchanges a couple of messages with the initiator process P_i. Thus, *O(n)* messages are sufficient for each time. Thus, considering k times, the message complexity of the algorithm is *O(kn)*.

Besides message complexity, another factor that must be considered as a performance measure is the number of pairwise comparisons of the checkpoints among the processes that is needed to be performed by any asynchronous checkpointing/recovery approach. This is done in order to determine a consistent global state of the system. Obviously larger the number of such comparisons, larger is the execution time of the recovery algorithm. This has been discussed in the previous subsection.

It may be noted that the number of such pairwise comparisons is also related to the number of times checkpoints are fetched from stable storage, i.e. the number of trips to the storage. The time spent on such trips may be substantial enough to affect to a good extent the speed of execution of any recovery algorithm. One possible solution may be to fetch a large number of checkpoints at a time. However, it may not be a good idea at all in many situations; for example, a process may end up in fetching too many when that many are not needed, or too little when more are needed. So, it becomes quite arbitrary about how many checkpoints should be fetched at a time. Therefore, it is wise to consider that a process will fetch one checkpoint at a time and in fact, this is true for all existing asynchronous checkpointing / recovery algorithms. In the following analysis we consider the fact that larger the number of pairwise comparisons of checkpoints, larger is the number of trips to stable storage, and therefore, larger is the execution time as a result.

In our analysis we will not consider complexity due to message size, as most related works including ours use control messages of reasonably small size and all these works differ mainly in terms of the number of comparisons, number of iterations, and the number of control message needed to determine a consistent global state. It may be noted that computing this number of comparisons is not very straightforward because it depends solely on the nature of the distributed computations. However, we give an approximate analysis which may not be very accurate; still it will offer a clear understanding of the advantages of our algorithm

---

Initiator process P_i:

Step 1: It asks every process P_j to send its V_j corresponding to its latest checkpoint $C_{j,r}$;
Step 2: It receives all V_j for $0 \leq j \leq n-1$;
Step 3: It computes $V_C = v_c^0\ v_c^1 \ldots v_c^j \ldots v_c^{n-1}$;
Step 4: It unicasts v_c^j to each P_j;
Step 5: It computes D_i by calculating $(R_i(r) - v_c^i)$;

if $D_i > 0$
It searches the list R_i till it finds the largest integer m (< r) that satisfies $R_i(r) - R_i(m) \geq D_i$. Then it sets its flag to 1 and considers V_i corresponding to its checkpoint $C_{i,m}$ (i.e. $C_{i,r}$ is replaced by $C_{i,m}$) for the next iteration;
/ Checkpoints $C_{i,r}$, $C_{i,r-1}$, ..., $C_{i,m+1}$ are skipped */*
else
It sets its flag to 0 and considers V_i at $C_{i,r}$ for the next iteration;

Step 6: It receives the flag and V_j from each process P_j;

if flag = 0 for each process P_j , $0 \leq j \leq n-1$
P_i asks each process P_j to restart the application program from its last checkpoint corresponding to which D_j ;

P_i resets its vector V_i to zero and list R_i to an empty list corresponding to its restarting checkpoint at which $D_i \leq 0$;
It restarts computation; */* its responsibility associated with the algorithm is finished */*

/ Globally consistent checkpoints belonging to the maximum consistent state are determined*/*

else
Control flows to Step 3;

Process P_j:

Step 1: P_j receives request from P_i;

if P_i has requested to restart
P_j resets its vector V_j to zero and list R_j to an empty list corresponding to its restarting checkpoint at which $D_j \leq 0$;
It restarts computation;
else
It sends V_j corresponding to its latest checkpoint $C_{j,r}$ to the initiator process P_i;

Step 2: It receives v_c^j from P_i;
Step 3: It computes D_j by calculating $(R_j(r) - v_c^j)$;
Step 4: if $D_j > 0$
It searches the list R_j till it finds the largest integer m (< r) that satisfies $R_j(r) - R_j(m) \geq D_j$. Then it sends a flag of 1 and V_j to P_i corresponding to its checkpoint $C_{j,m}$ (i.e. $C_{j,r}$ is replaced by $C_{j,m}$);
/ Checkpoints $C_{j,r}$, $C_{j,r-1}$, ..., $C_{j,m+1}$ are skipped */*
else
It sends a flag of 0 and V_j at $C_{j,r}$ to the initiator process P_i;

Figure 2: The responsibilities of each participating process P_j and the initiator process P_i

over some other noted asynchronous checkpointing / recovery approaches [14], [16]. It is stated below.

Let the system consist of n processes. For simplicity we assume that after a failure occurs and the system recovers from it, each process will skip on an average its latest (r-1) checkpoints to restart its computation. Thus a process P_j will skip its latest (r-1) checkpoints $C_{j,m+2}$, …. , $C_{j,r+m}$. We also assume that the set $\{C_{0,m+1}, C_{1,m+1}, \ldots, C_{n-1,m+1}\}$ represents the globally consistent checkpoint (maximum consistent state) of the system and our algorithm will determine it in k number of iterations. In this simple model, we consider a recovery approach associated with asynchronous checkpointing scheme in which the pairwise comparisons to determine checkpoints' consistency involves first the checkpoints of the set $\{C_{0,m+r}, \ldots. , C_{n-1,m+r}\}$, followed by the set $\{C_{0,m+r-1}, \ldots. , C_{n-1,m+r-1}\}$, … and so on, and finally the set $\{C_{0,m+1}, \ldots., C_{n-1,m+1}\}$which is the globally consistent state. Therefore, the total number of comparisons is given by [r x{n(n-1)}/2]. Note that this may not be the exact way to perform the comparisons in a particular case; still it offers a clear view of how complex it can be. In general, a checkpoint(s) in one set may also have to be compared with a checkpoint(s) in another set. On the other hand, not necessarily all checkpoints in a set may be needed to be pairwise compared. It depends on the nature of the distributed computations. So the actual number of comparisons may be larger or smaller than the number [r× {n(n-1)}/2]. Anyway, it is clear that this number is much larger than the total number of comparisons k×n, offered by our approach, where n is the number of parallel comparisons to test if $D_j > 0$ in each iteration and $1 \leq k \leq r$. Observe that in the worst case, the number of comparisons of the proposed approach may become [r× {n(n-1)}/2]. Below we have compared the performance of our approach with the approaches in [14], [16].

6.3.1 Comparison with Ohara et. al. [14]

Ohara et al. [14] have proposed an asynchronous approach for finding a recovery line where a given checkpoint is the earliest. All the local checkpoints which are just behind a given checkpoint are initially assumed to form a consistent global checkpoint. In this algorithm, happened-before relations are checked for every coupled local checkpoints belonging to an ordered global checkpoint set. If there exists any happened-before relation, it replaces a local checkpoint with a successive local checkpoint of the same process. The algorithm may end by either finding a recovery line or running out of local checkpoints to be replaced. This leads to exhaustive comparisons of happened before relations for every coupled local checkpoints. The number of such comparisons is approximately [r× {n(n-1)}/2] as calculated earlier. In our algorithm, it skips the checkpoints that do not belong to the set of the globally consistent checkpoints. Thus, our algorithm reduces to a good extent unnecessary pairwise comparisons of the checkpoints to determine global consistent checkpoint of the system. Performance comparison of the above mentioned approach [14] and our approach is shown in Fig. 4.

Fig. 3 illustrates how the number of comparisons is affected with the increase in the average number of checkpoints per process (r) in the asynchronous approach [14] and in our approach. Fig. 4 shows the variation of the number of comparisons with the increase in the number of processes (n). Both figures highlight the advantages offered by our approach, i.e. considerable amount of reduction in the number of comparisons in our approach. It helps the processes to restart their computation related to the distributed application much faster after the system recovers from a failure.

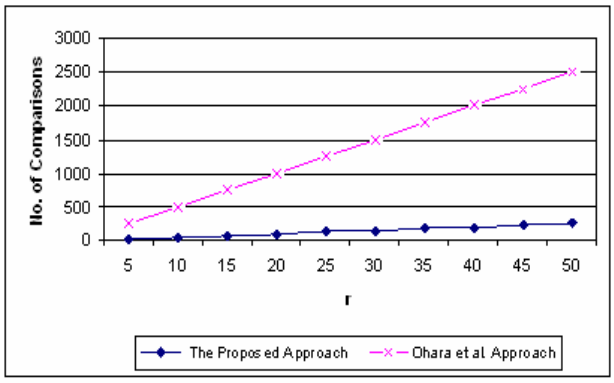

Figure 3: Number of comparisons vs. the average number of checkpoints per process (r).

6.3.2 Comparison with Venkatesan et. al. [16]

Venkatesan and Juang [16] presented an asynchronous checkpointing algorithm where each process takes checkpoints independently and keeps track of the number of messages it has sent to other processes as well as the number of messages it has received from other processes. The existence of orphan messages is discovered by comparing the number of messages sent and received. The algorithm is initiated by the process when a failure occurs or when it learns about process failure.

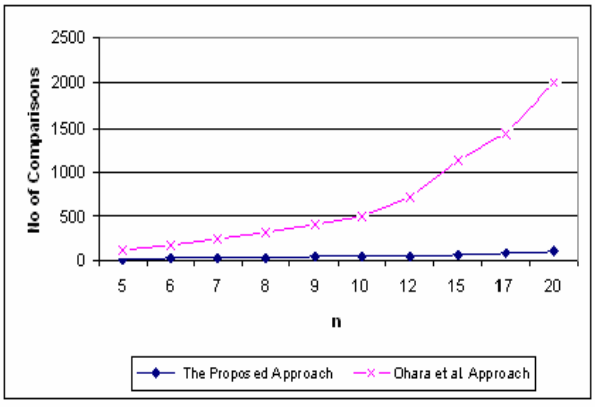

Figure 4: Number of comparisons vs. the number of processes (n).

During its each iteration, a process needs to compare the number of messages received by it and the actual number of messages sent by the other process, at each of its checkpoints starting from the recent one. The received vectors corresponding to all the checkpoints including the current one and the one where next iteration should start, need to be fetched from the storage in order to decide the checkpoint for the next iteration to start with. It means that the number of trips to the storage for fetching the information related to the received message (for the purpose of comparison) will be equal to the number of checkpoints starting from the current checkpoint all the way to the checkpoint where the next iteration should start.

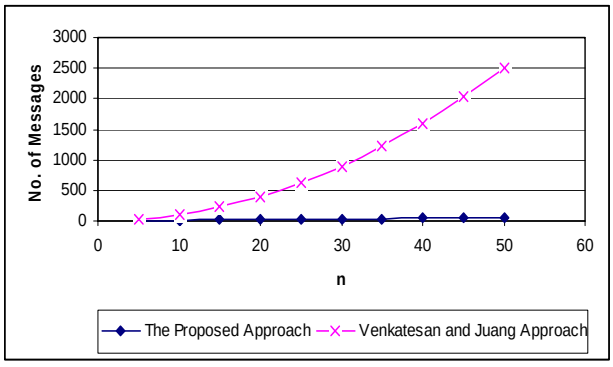

Figure 5: Number of control messages vs. the number of processes (n).

In our algorithm, the decision about the checkpoint at which the next iteration should start is based on the R-vector at the recent checkpoint only. This algorithm skips checkpoints that do not belong to the set of the globally consistent checkpoints by examining this R-vector only. Therefore, in order to determine the checkpoint for the next iteration to start with, the number of trips to the storage is only one per iteration. This means that the total number of trips to complete the execution of our algorithm is reduced to a good extent compared to that in [16]. We now compare the two algorithms based on the number of control messages needed to execute the respective algorithms.

In [16], in each iteration, for an n-process system n(n-1) messages are exchanged among the processes. Thus, $O(n^2)$ messages are exchanged in each iteration. In our algorithm, 3(n-1) messages are exchanged in each iteration. Thus, $O(n)$ messages are sufficient in each iteration in our algorithm where n is the number of processes in the system. Fig. 5 shows the message complexity comparison of the two algorithms with the increase in the number of processes. This figure clearly shows the advantage offered by our algorithm over the one in [16].

7 Further Enhancement

We have seen that the linear list R_j maintained by a process P_j increases dynamically. If the application program has large execution time and there is seldom any failure during its execution, the length of the lists may become too large; thereby it may consume considerable amount of memory. To solve this problem, i.e. to keep the list from growing too much we will propose a simple solution in this section. The following operation is needed in the implementation of the idea.

We define the subtraction operation on two vectors V_j of process P_j at its two checkpoints $C_{j,m}$ and $C_{j,s}$ with (s > m) as follows:

V_j at $C_{j,s} - V_j$ at $C_{j,m}$ = [($v_{j,0}$ at $C_{j,s} - v_{j,0}$ at $C_{j,m}$), … , ($v_{j,n-1}$ at $C_{j,s} - v_{j,n-1}$ at $C_{j,m}$)] = [($v_{j,p}$ at $C_{j,s} - v_{j,p}$ at $C_{j,m}$)]
for $0 \le p \le n-1$

We now state the basic idea to keep the growing lengths of the lists in control. This idea has been used in designing the enhanced recovery algorithm stated later in this section. It may be noted that the recovery algorithm stated earlier does not consider the use of this idea.

In absence of any failure an algorithm runs periodically (say the time period is T which should be much larger than the time period of any individual process) to put a limit on the length of the R-vector. The lengths of the lists (R-vectors) may then be limited by the number of checkpoints taken by the processes during the time interval (T) between two successive executions of the algorithm. Besides in doing so, this also advances the recovery line in the event that a recent recovery line exists other than the one found during the previous execution of the algorithm. In effect, the number of comparisons of the checkpoints to determine a recent consistent state may also drastically reduce since there is a possibility that the algorithm will consider in a particular run only the checkpoints which the processes take during the interval T. Therefore, this enhanced algorithm, in general, may take much less time to complete its execution compared to Algorithm Recovery. Also note that at the completion of the l^{th} execution of the algorithm a process P_j will have in stable storage only its recent globally consistent checkpoint, say $C_{j,m}$ and any other checkpoint (s) it has taken thereafter and prior to the start of the l^{th} periodic execution of the algorithm.

In describing the following two rules for updating the lists R_j and the vectors V_j of a process P_j we have assumed that the latest globally consistent checkpoint of process P_j is $C_{j,m}$ as determined by the l^{th} execution of the algorithm and it has taken (k-m) more checkpoints thereafter and prior to the start of the l^{th} periodic execution of the algorithm.

Rule 1: Updated R_j at $C_{j,m}$ = {} and updated V_j at $C_{j,m}$ = [00…0]

Rule 2: Updated R_j at $C_{j,s}$ for ($m+1 \le s \le k$) = [($R_j(m+1) - R_j(m)$), … , ($R_j(s) - R_j(m)$)], and
Updated V_j at each $C_{j,s}$ = [($v_{j,p}$ at $C_{j,s} - v_{j,p}$ at $C_{j,m}$)]
for $0 \le p \le n-1$

When we implement the above idea of reducing the lengths of the lists, either of the following two approaches can be adopted:

Approach 1: When a failure occurs and the system recovers from the failure, the algorithm is run again in

spite of its periodic execution, with the hope that a recent (maximum) consistent state may be found which is not identical to the one determined by its last periodical execution. In such a situation the time to complete the application will be less because of the advancement of the recovery line.

On the other hand, if such a situation as mentioned above does not exist, the algorithm will output the same consistent state as determined in its last periodic execution. In this case, however, the application will take an additional amount of time equal to the execution time of the algorithm for its completion.

Approach 2: After the system recovers from a failure all processes restart from their respective globally consistent checkpoints which have already been determined by the algorithm's last periodic execution prior to the occurrence of the failure. The recovery becomes as simple as that in a synchronous approach. However, since this approach does not look for the possible existence of a recent consistent state other than the already existing one, therefore the time to complete the application may increase.

Observe that irrespective of which approach is followed, the next periodic execution of the algorithm will occur T time units after the system restarts. About when to apply a specific rule, Rules 1 and 2 will be implemented when the algorithm runs periodically in absence of any failure. Rule 1 is also implemented when determination of a consistent global state of the system is needed after the system recovers from a failure (Approach 1). In the following algorithm we have considered a combination of the two approaches.

For the selection of an initiator process for running the algorithm periodically, we consider that each process P_i maintains a local CLK_i variable which is incremented at periodic time interval T. It also maintains a local counter denoted as $counter_i$, initially set to 0 and is incremented by process P_i during its turn to initiate the recovery algorithm. Thus, a process on its own determines if it is its turn to initiate the execution of the algorithm. In this context, observe that the set of GCCs is unique and is independent of the initiator process. We state below how a process P_i does it before we formally state the algorithm:

Selection of an initiator process:
At each process P_i ($0 \leq i \leq n-1$):
If $CLK_i = (i+(counteri*n))*T$
counteri= counteri+1;
/*When its turn to initiate the recovery algorithm, i.e., P_i becomes the initiator*/

Algorithm Recovery – Enhanced:
Input: Given the latest n checkpoints, one for each process P_j , $0 \leq j \leq n-1$, for an n process system and the corresponding vectors V_j and lists R_j at these n checkpoints.
Output: A set of globally consistent checkpoints (maximum consistent state of the system).

The responsibilities of the initiator process P_i and each participating process P_j are stated in Fig. 6.

An example: Consider the system as shown in Fig. 7. Ignore the presence of the failure 'f' for the time being. Suppose that the periodic execution of algorithm starts immediately after processes P_1 and P_3 take their respective checkpoints $C_{1,5}$ and $C_{3,2}$. The algorithm determines the latest consistent global checkpoint of the system. It is $\{C_{1,2}, C_{2,1}, C_{3,1}\}$.

The two rules are applied to update the lists R_1, R_2, and R_3, and the vectors V_1, V_2, and V_3 at the checkpoints of processes P_1, P_2, and P_3 starting from their respective latest globally consistent checkpoints, which are namely $C_{1,2}$, $C_{2,1}$, $C_{3,1}$. The system with the updated lists and vectors is shown in Fig. 8. The checkpoints shown in Fig. 8 are the only ones saved in stable storage.

Now assume that a failure 'f' has occurred. Therefore the algorithm determines the consistent global checkpoint of the system, which is $\{C_{1,2}, C_{2,1}, C_{3,1}\}$ and applies only Rule 1 to reset the vectors to zero and to make the lists empty at the respective GCCs of the three processes.

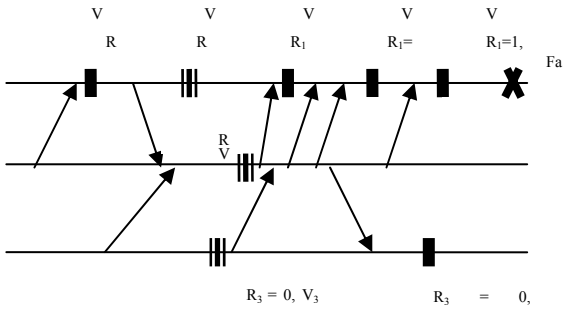

Figure 7: Before the execution of the algorithm

The system in this situation is shown in Fig. 9. The three respective consistent checkpoints are the only ones saved in the stable storage at this time.

Note that the consistent global state remains the same (see Figs. 8 and 9). This is the situation when time to complete the application program increases by an amount equal to the time to execute the recovery algorithm. This has been pointed out earlier in the description of Approach 1. However, this will not happen if only Approach 2 is followed for recovery.

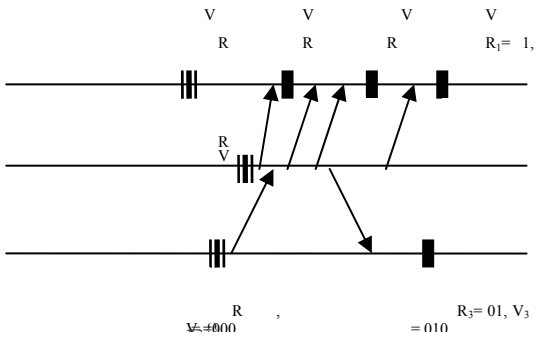

Figure 8: After the execution of the algorithm in absence of any failure

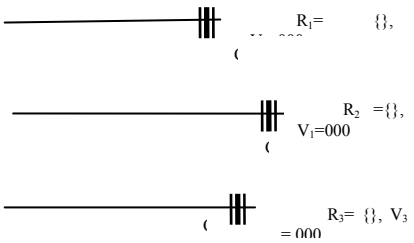

Figure 9: The system restarts from its consistent global state $\{C_{1,2},C_{2,1},C_{3,1}\}$ after recovery.

7.1 Comparison with [11] and [13]

Gupta et al. [11] have proposed a roll-forward hybrid checkpointing / recovery scheme using basic checkpoints. The direct dependency concept used in the communication-induced checkpointing scheme has been applied to basic checkpoints to design a simple algorithm to find a consistent global checkpoint. They have used the concept of forced checkpoints that ensures a small re-execution time after recovery from a failure. This scheme has the advantages of simple recovery as in synchronous approach and simple way to create checkpoints like in asynchronous approach.

Our proposed approach (enhanced version) is not a hybrid approach. It runs periodically only to put a limit on the size of the R-vectors. This is the primary objective of the enhanced approach. In doing so it may come out with a recent recovery line that is different from the one found during the last execution of the algorithm. Thus, effectively as mentioned earlier, even though the proposed algorithm is not a hybrid one, still as in [11] it may reduce drastically the number of comparisons needed to identify a recovery line, as well as it may limit the domino effect by the time period T, based on the message communication pattern among the processes.

Our proposed approach is quite different from the work in [13] in that in our approach processes take checkpoints completely independently based on their individual time periods that are different for different processes. In [13], processes take checkpoints with the same time periods and they make sure that there is no orphan message between any two i^{th} checkpoints of two processes. Therefore, it is more of a synchronous approach than an asynchronous approach, where as our approach is purely an asynchronous approach.

8 Conclusions

In this paper we have presented an efficient recovery algorithm for distributed systems. Asynchronous checkpointing scheme has been considered because of its simplicity in taking checkpoints. The main feature of the recovery algorithm is that to determine a maximum consistent state, the algorithm in its each iteration does not need to compare all the vectors at all the checkpoints of the processes. In its each iteration the algorithm identifies and skips those checkpoints that can not belong to the set of the globally consistent checkpoints. It not only reduces the computational overhead to a good extent, but also the number of trips to the stable storage for fetching checkpoints is reduced compared to the works in [14] and [16], and as a result its execution becomes even faster. In this context, it may be noted that in any algorithm that uses asynchronous checkpointing, there is always some computational time wasted to create process checkpoints that later do not belong to CGS and this problem can not be avoided. This is true for our proposed algorithms as well. Besides, it is executed simultaneously by all participating processes while determining a maximum consistent state. It further contributes to its speed of execution. We have also proposed a simple enhanced asynchronous recovery scheme to control the dynamically growing length of the lists. In effect, the number of comparisons of the checkpoints to determine a recent consistent state may also drastically reduce and based on the communication pattern among the processes it may limit the domino effect by the time period T. Even though we do not apply any hybrid checkpointing scheme [11], still this approach offers the option to achieve a recovery scheme which is as simple as the approach proposed in [11]. In this context, it may be noted that if the system model changes such that order of the messages sent through the channel cannot be preserved, it will adversely affect the processing time, because a process must wait to receive message m_1 before processing its already received message m_2. Here, we have assumed that the proper order is m_1 followed by m_2.

Our future work is directed at the new challenging area of designing recovery schemes for cluster federation computing environment in which different clusters may adopt different ways for checkpointing, for example, some may apply coordinated approach, where as other may apply asynchronous approach [18], [19].

```
Initiator process Pi:
Step 1:     if the algorithm is executed on failure
                recovery_on_failure = 1;
            else  recovery_on_failure = 0;
            It asks every process Pj (j ≠ i) to send its Vj corresponding to its latest checkpoint Cj,r;

Step 2:     It receives all Vj for 0 ≤ j ≤ n-1;
Step 3:     It computes Vc = vc^0 vc^1 … vc^j … vc^n-1;
Step 4:     It unicasts vc^j to each Pj, for j ≠ i;
Step 5:     It computes Di by calculating (Ri(r) - vc^i);
            if Di> 0
                It searches the list Ri till it finds the largest integer m (< r) that satisfies Ri(r) – Ri(m) ≥ Di. Then it
                sets its flag to 1 and considers Vi corresponding to its checkpoint Ci, m (i.e. Ci,r is replaced by Ci,m ) for
                the next iteration;

                                                /* Checkpoints Ci,r, Ci,r-1, …., Ci, m+1 are skipped */
            else
                It sets its flag to 0 and considers Vi at Ci, r for the next iteration;
Step 6:     It receives the flag and Vj from each process Pj;
            if flag = 0 for each process Pj , 0 ≤ j ≤ n-1

            /*Globally consistent checkpoints belonging to the maximum consistent state are determined */

                if recovery_on_failure = 1
                   Pi asks each process Pj to restart the application program from its last checkpoint
                   corresponding to which Dj ;
                   Pi implements Rule 1 corresponding to its restarting checkpoint at which Di ≤ 0;
                   It restarts computation from the restarting checkpoint (its GCC);
                else
                   Pi asks each process Pj to continue its normal computation from its latest checkpoint;
                   Pi implements Rule 2 followed by Rule 1;  /* Periodic determination of GCCs is done */
                   It continues its normal computation from its latest checkpoint;
            else
                Control flows to Step 3;
-------------------------------------------------------------------------------------------------------------------------
Process Pj:
Step 1:     Pj receives request from Pi;

                if Pi  has requested to restart
                                                /* The system restarts after recovery from a failure*/

                Pj implements Rule 1 corresponding to its restarting checkpoint at which Dj ≤ 0;
                It restarts computation from the restarting checkpoint (its GCC);
            else if
                Pi has requested to continue with the application program
                Pj implements Rule 2 followed by Rule 1;
                                                /* Periodic determination of GCC is done */
                It continues its normal computation from its latest checkpoint;
            else
                It sends Vj corresponding to its latest checkpoint Cj,r to the initiator process Pi;
Step 2:     It receives vc^j from Pi;
Step 3:     It computes Dj by calculating (Rj(r) - vc^j);
Step 4:     if Dj > 0
                It searches the list Rj till it finds the largest integer m (< r) that satisfies Rj(r) - Rj(m) ≥Dj. Then it
                sends a flag of 1 and Vj to Pi corresponding to its checkpoint Cj, m (i.e. Cj,r is  replaced by Cj,m );

                                                /* Checkpoints Cj,r, Cj,r-1, …., Cj, m+1 are skipped */
            else
                It sends a flag of 0 and Vj at Cj,r to the initiator process Pi;
```

Figure 6: The responsibilities of the initiator process P_i and each participating process P_j for the enhanced algorithm.

9 References

[1] R. Koo and S. Toueg, "Checkpointing and Rollback-Recovery for Distributed Systems", IEEE trans. Software Engineering, vol. SE-13, no. 1, pp. 23-31, Jan 1987.

[2] Y. M. Wang, A. Lowry, and W. K. Fuchs, "Consistent Global Checkpoints Based on Direct Dependency Tracking", Information Processing Letters, vol. 50, no. 4, pp. 223-230, May 1994.

[3] K. M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems", ACM Trans. Computing Systems, vol.3, no. 1, pp. 63-75, Feb. 1985.

[4] Y. Wang, "Consistent Global Checkpoints that Contain a Given Set of Local Checkpoints", IEEE Trans. Computers, vol. 46, no. 4, pp. 456-468, April 1997.

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

[6] S. Venkatesan, T. Juang, and S. Alagar, "Optimistic Crash Recovery Without Changing Application Messages", IEEE Trans. Parallel and Distributed Systems, vol. 8, no.3, pp. 263-271, March 1997.

[7] R. Baldoni, F. Quaglai, and P. Fornara, "An Index-Based Checkpointing Algorithm for Autonomous Distributed Systems", IEEE Trans. Parallel and Distributed System, vol. 10, no.2, pp.181-192, Feb. 1999.

[8] J. Tsai, S. -Y. Kuo, and Y. -M. Wang, "Theoretical Analysis for Communication-Induced Checkpointing Protocols with Rollback Dependency Trackability", IEEE Trans. Parallel and Distributed Systems, vol. 9, no. 10, pp. 963-971, Oct. 1998.

[9] G. Cao and M. Singhal, " On Coordinated Checkpointing in Distributed Systems, IEEE Trans. Parallel and Distributed Systems ", vol. 9, no.12, pp. 1213-1225, Dec.1998.

[10] P. Jalote, Fault Tolerance in Distributed Systems, PTR Prentice Hall, (1994), Addison-Wesley, (1998).

[11] B. Gupta, S.K. Banerjee and B. Liu, "Design of new roll-forward recovery approach for distributed systems", IEE Proc. Computers and Digital Techniques, Volume 149, Issue 3, pp. 105-112, May 2002.

[12] D. Manivannan and M. Singhal, "Quasi-synchronous checkpointing: Models, characterization, and classification", Parallel and Distributed Systems, IEEE Transactions on Volume 10, Issue 7, pp. 703– 713, July 1999.

[13] D. Manivannan, and M. Singhal, "Asynchronous recovery without using vector timestamps", Journal of Parallel and Distributed Computing, Volume 62, Issue 62 pp. 1695-1728, Dec 2002.

[14] M. Ohara., M. Arai., S. Fukumoto., and K. Iwasaki.,"Finding a Recovery Line in Uncoordinated Checkpointing", Proceedings 24th International Conference on Distributed Computing Systems Workshops (ICDCSW'04), pp. 628 – 633, 2004.

[15] R. H. B. Netzer and J. Xu, "Necessary and Sufficient Conditions for Consistent Global Snapshots", IEEE Trans. Parallel and Distributed Systems, vol. 6, no. 2, pp. 165-169, Feb. 1995.

[16] T. Juang and S. Venkatesan, "Crash Recovery with Little Overhead", Proc. 11th International Conference on Distributed Computing Systems, pp. 454-461, May 1991.

[17] B. Gupta, Y. Yang, S. Rahimi, and A. Vemuri, "A High-Performance Recovery Algorithm for Distributed Systems", Proc. 21st International Conference on Computers and Their Applications, pp. 283-288, Seattle, March 2006.

[18] S. Monnet, C. Morin, R. Badrinath, "Hybrid Checkpointing for Parallel Applications in cluster Federations", Proc. 4th IEEE/ACM International Symposium on Cluster Computing and the Grid, Chicago, IL, USA, pp. 773-782, April 2004.

[19] J. Cao, Y. Chen, K. Zhang and Y. He, "Checkpointing in Hybrid Distributed Systems", Proc.7th International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN'04), pp. 136-141, May 2004.