

A Fast And Efficient Non-Blocking Coordinated Checkpointing Approach For Distributed Systems

B. Gupta and S. Rahimi
Computer Science Department
Southern Illinois University
Carbondale, IL 62901 USA

Abstract - In this paper, we have presented an efficient non-blocking coordinated checkpointing algorithm for distributed systems. The distinct advantages of the proposed algorithm are the following. It produces a consistent set of checkpoints, without the overhead of taking temporary checkpoints; the algorithm also makes sure that only few processes are required to take checkpoints in its any execution; it uses very few control messages, and the participating processes are interrupted fewer number of times when compared to some noted related works. These distinct advantages make the algorithm fast and efficient.

Keywords: Coordinated Checkpointing, Non- blocking approach, Distributed systems.

1. Introduction

Systems may fail from time to time, despite how well they are designed. Checkpointing and rollback- recovery [1],[2],[7] is one of the widely used techniques that allow systems to progress in spite of a failure. A checkpoint is a snapshot of the current state of the process stored in stable storage. This information is used to reconstruct the process state if the contents of the volatile storage are lost due to process failure. A set of checkpoints, with one checkpoint for every process is said to be consistent global checkpointing state (CGS) if it does not have any orphan messages.

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

In coordinated checkpointing, all processes synchronize through control messages before taking checkpoints. These synchronization messages contribute to extra overhead but make the system free from domino effect. Coordinated checkpointing algorithms are of two types: (a) blocking [7] and (b) non-blocking [3],[5],[8]. Blocking algorithms force all relevant processes in the system to block their computation during checkpointing latency and hence degrade system performance. In non-blocking algorithms application processes are not blocked when checkpoints are being taken.

Problem Formulation: Most research in the area of coordinated checkpointing concentrates on non-blocking. In [3] and [4] the authors have proposed non-blocking coordinated checkpointing algorithms that require only a minimum set of processes to take checkpoints at any instant of time. In the earlier works though processes need not block, but all processes need to take checkpoints which are unnecessary. To avoid this, in [3] the authors introduced the concept of mutable checkpoints. They are neither temporary nor permanent checkpoints. To decide whether to convert a mutable checkpoint to a permanent checkpoint is a complex one.

In [4] the authors have proposed an algorithm which is non-blocking and also coordinated. It is a five step algorithm where first the initiator process requests the dependency vectors of all the processes. The initiator receives the dependency vectors from all the processes, from which it creates the minimum set of processes which need to take checkpoints. Then the initiator takes its own tentative checkpoint and sends checkpoint request to the processes in the minimum set. After some time the initiator receives responses from all the processes that took the tentative checkpoints. It then sends the commit or abort message to all processes. All the processes that did not send or receive messages will not participate in the checkpointing algorithm. Hence at any instant of time only minimum number of processes take checkpoint.

In our work we present a non blocking coordinated checkpointing approach to find a globally consistent state of the system. It is a three step (phase) algorithm. Our algorithm requires very few control messages and few interrupts to each participating process. The algorithm also eliminates the overhead of taking temporary/mutable checkpoints. In this algorithm at any instant of time any one process can initiate the checkpointing algorithm.

This paper is organized as follows. In Section 2 we present the system model. In Section 3 the relevant data structures and the algorithm is presented along with its advantages. Section 4 draws the conclusion.

2. System Model

The distributed system that we have considered has the following characteristics: Processes do not share memory and they communicate via messages sent through the channels. All channels are fault-prone, that is they can lose messages. However, the order of messages is preserved by some end-to-end transmission protocol.

3. Data Structures

Consider a set of n processes, $\{P_1, P_2, \dots, P_n\}$ involved in the execution of a distributed algorithm. Each process P_i maintains a dependency vector DV_i of size n which is initially empty. And $DV_i[j]$ is set to 1 when P_i receives at least one message from P_j . It is reset to 0 again when process P_i takes a checkpoint. Each process P_i also maintains a checkpoint sequence number csn_i , where csn_i represents current checkpointing interval of process P_i . The i^{th} checkpointing interval of a process denotes all the computation performed between its i^{th} and $(i+1)^{th}$ checkpoint, including the i^{th} checkpoint but not the $(i+1)^{th}$ checkpoint. The csn_i is initially set to 1 and is incremented when process P_i takes a checkpoint.

In this approach we assume that at any instant of time only one process can initiate the checkpointing algorithm. This process is known as the initiator process.

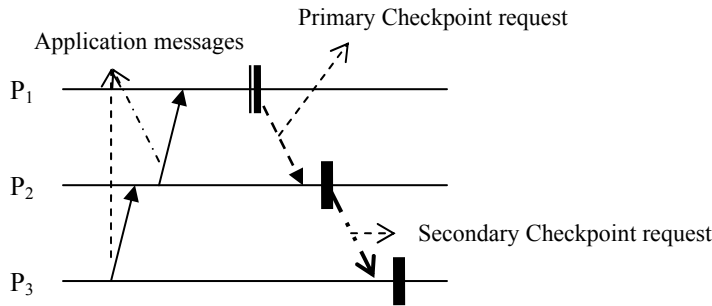


Fig. 1 Control message exchanges in our approach.

In our proposed algorithm we assume primary and secondary checkpoint request exchanges between the initiator process and the system of $n-1$ processes. A primary checkpoint request is denoted by R_i ($i = csn_j$) where i is the current checkpoint sequence number of process P_j that initiates the checkpointing algorithm. It is sent by the initiator process P_j to all dependent processes asking them to take their respective checkpoints. A secondary checkpoint request denoted by R_{si} is sent from a dependent process P_m to a process P_n which is dependent on P_m to take a checkpoint. R_{si} also means to its receiver process that i is the current checkpoint sequence number of the sender process. The control message exchange is explained with an illustration shown in Fig. 1. Consider a distributed system with three processes P_1 , P_2 , and P_3 . We assume that P_1 initiates the checkpointing algorithm. Initially assume that P_1 takes a checkpoint and sends a primary checkpoint request to P_2 , asking it to take a checkpoint as it is directly dependent on P_1 . P_2 takes a checkpoint after it receives the primary checkpoint request. After P_2 has taken its checkpoint it sends a secondary checkpoint request to P_3 as P_3 is dependent on P_2 , because P_2 has already received an application message from P_3 and has taken a checkpoint thereafter.

An application message is represented by $M_{i,x}$, which means that it is the x^{th} message sent by process P_i . The checkpoint $C_{i,j}$ represents the j^{th} checkpoint taken by P_i . We now state the situation in general when a process P_i needs to take a checkpoint.

In our approach a process P_i takes a checkpoint if any of the following events occurs:

1. if P_i is the initiator
2. if it receives a primary checkpoint request from the initiator
3. if it receives a secondary checkpoint request
4. if it receives an application message piggybacked with the checkpoint sequence number, but has not yet received a request message.

Before the formal statement of the algorithm, the working principle of the algorithm is illustrated with an example. For a clear understanding, different possible scenarios are discussed. Each process P_i is supposed to piggyback its current checkpoint sequence number with every first outgoing application message to a process after taking a checkpoint.

3.1 An illustration

The behavior of each process is explained with the help of the following illustration. Unless otherwise mentioned a checkpoint request represents either a primary request or a secondary request or an application message with piggybacked information which forces a checkpoint to be taken at the receiving process.

Consider seven processes P_1, P_2, \dots, P_7 in a distributed system. Each primary checkpoint request is represented by R_{csn} and a secondary checkpoint request is represented by $R_{s\ csn}$. ($csn =$ checkpoint sequence number of initiator).

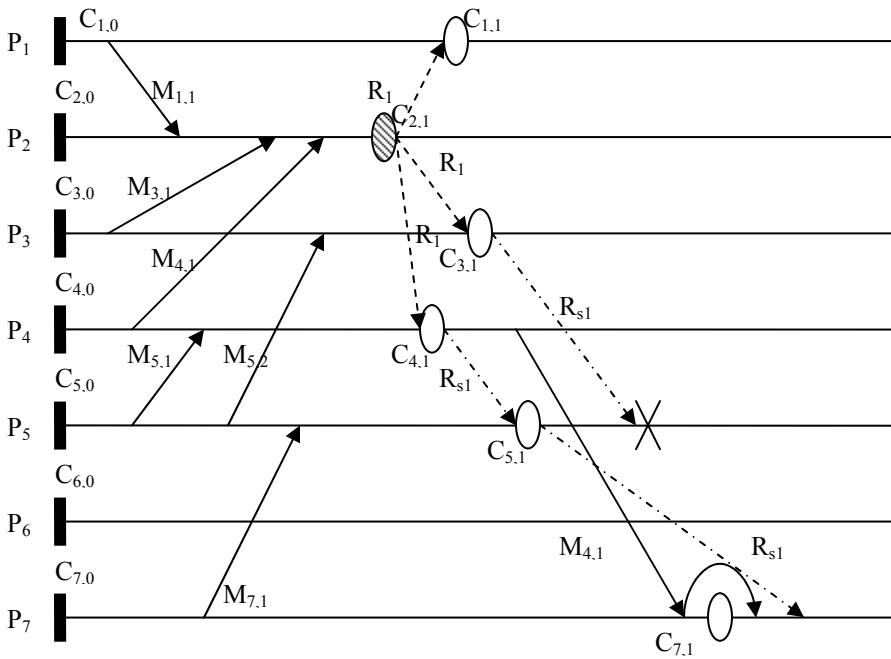


Fig. 2 Different situations for taking checkpoints

Consider a distributed system as shown in the Fig. 2. Assume that process P_2 initiates the checkpointing algorithm. First process P_2 takes a checkpoint $C_{2,1}$. It then checks its dependency vector $DV_2[]$ which is $\{1,0,1,1,0,0,0\}$. This means that process P_2 has received at least one message from processes P_1, P_3 , and P_4 , and since P_2 has already taken checkpoint $C_{2,1}$ these messages become orphan. Therefore P_2 sends primary checkpoint request R_1 ($csn_2 = 1$) to P_1, P_3, P_4 . After sending the primary checkpoint request process P_2 increments its checkpoint sequence number csn_2 to 2. P_2 terminates its execution of the algorithm and continues normal computation.

On receiving the primary checkpoint request R_1 from P_2 , process P_3 first takes a checkpoint $C_{3,1}$ and then it checks its own dependency vector $DV_3[]$ which is $\{0,0,0,0,1,0,0\}$. Therefore process P_3 sends a secondary checkpoint request R_{s1} to process P_5 . Then its checkpoint sequence number csn_3 is incremented to 2. Similarly processes P_1 and P_4 first take checkpoints $C_{1,1}$ and $C_{4,1}$ respectively, then each process checks its dependency vector to find the dependent processes. Process P_1 finds that its dependency vector $DV_1[]$ is null. Hence it increments its checkpoint sequence number to 2, and continues normal execution. Process P_4 finds that it has received a message

from process P_5 . Hence P_4 sends a secondary checkpoint request R_{s1} to process P_5 . It then increments its checkpoint sequence number csn_4 to 2, and continues normal execution.

At process P_5 let us assume that the secondary checkpoint request R_{s1} sent by process P_4 reaches before the secondary checkpoint request sent by process P_3 . On receiving the secondary checkpoint request R_{s1} from process P_4 , P_5 checks its own checkpoint sequence number csn_5 with that of the received checkpoint sequence number. P_5 finds that its current checkpoint sequence number ($csn_5 = 1$) is not greater than the received checkpoint sequence number which is equal to 1. Hence it decides to take a checkpoint and takes checkpoint $C_{5,1}$. After taking the checkpoint it checks its dependency vector $DV_5[]$ and decides that process P_7 has sent a message to it. Hence it sends a secondary checkpoint request R_{s1} to P_7 . After sending the request it increments its checkpoint sequence number csn_5 from 1 to 2. Assume that later process P_5 receives the secondary checkpoint request sent by process P_3 . As soon as process P_5 receives the checkpoint request it compares its current checkpoint sequence number csn_5 with the received checkpoint sequence number. It finds that its current checkpoint sequence number ($csn_5 = 2$) is greater than the received checkpoint sequence number which is 1. Hence it discards the checkpoint request. The above discussion takes care of the first three situations about when a process takes a checkpoint. Below, we consider the fourth situation.

Suppose that the process P_4 after taking the checkpoint continues normal execution and sends an application message $M_{4,1}$ to process P_7 . Since the application message is the first application message to process P_7 from P_4 after taking the checkpoint, it is piggybacked with the current checkpoint sequence number (csn_4) of process P_4 which is 2. Process P_7 on receiving the application message piggybacked with the checkpoint sequence number; compares its current checkpoint sequence number csn_7 with the received checkpoint sequence number. It finds that the received checkpoint sequence number is equal to 2 and is greater than its current checkpoint sequence number (csn_7) which is equal to 1. Therefore process P_7 decides to take a checkpoint before processing the application message $M_{4,1}$. P_7 then takes checkpoint $C_{7,1}$ and increments its checkpoint sequence number to 2 and then processes the application message $M_{4,1}$.

Eventually process P_7 also receives the secondary checkpoint request sent by process P_5 . P_7 first compares its current checkpoint sequence number with the received checkpoint sequence number which is 1. It finds that its current checkpoint sequence number is greater than the received checkpoint request. Hence P_7 discards the secondary checkpoint request as it has already taken checkpoint in that checkpointing interval.

The above discussion leads to the following observation.

Observation 1: If a process P_k has not yet participated in the current execution of the checkpointing algorithm, it takes a checkpoint at the first occurrence of an interrupt caused by a checkpoint request which is either a primary checkpoint request or a secondary checkpoint request or a piggybacked application message and later ignores any other checkpointing request, if any associated with the current execution of the checkpointing algorithm.

We now describe the avalanche effect that may occur in a coordinated checkpointing approach, based on some typical communication pattern among the processes.

Avalanche Effect: Consider the following situation: suppose process P_i initiates the coordinated checkpointing scheme. It checks its dependency vector and sends request to all processes that are directly dependent on it. Suppose P_j receives the primary request from P_i since it is directly dependent on P_i ; P_j checks its own dependency vector and sends a secondary request to P_k and takes a checkpoint. P_k in turn checks its dependency vector and sends a request to P_r and takes a checkpoint and it goes on in such a way that P_i gets a secondary request from some process P_s , because P_i is dependent on P_s and so it takes a checkpoint again, and sends a secondary request to another process looking at its dependency vector. If this continues then the checkpointing scheme can never terminate. This phenomenon is known as avalanche effect.

Claim 1: Avalanche Effect does not occur in our approach.

Proof: Assume that avalanche effect is possible in our approach. Without any loss of generality, let us assume that process P_i has sent a primary checkpoint request to P_k ; P_k after checking its dependency vector $DV_k[]$ sends a secondary request to P_r , and P_r acts similarly and sends request to P_q . Following the same way finally some process

P_s sends a secondary checkpoint request to P_i looking at its dependency vector $DV_s[]$. Therefore it appears that $P_i, P_k, P_r, P_q, \dots, P_s, P_i$ form a loop of requests.

Since process P_i already has taken its checkpoint, so according to Observation 1, it discards the request. Hence this chain of requests, can have a maximum length of n where n is the total number of processes in the system. This shows that once a process in the chain finishes its participation in the current checkpointing algorithm, it will not participate in the current checkpointing algorithm again. Hence the assumption that avalanche effect is possible, is not valid in our approach. ■

In the following algorithm we have considered all the cases about when a process takes a checkpoint.

3.2 Algorithm Non_Blocking

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

Initiator process P_i

```

Step 1: take a checkpoint, check the dependency vector  $DV_i[]$ ;
Step 2: when  $DV_i[k] = 1$  for  $1 \leq k \leq n$ 
        Send a Primary request- $R_n$  to process  $P_k$ ;
        /* checks the dependency vector and multicasts a checkpoint request */
Step 3: increment the checkpoint sequence number  $csn_i$ ;
Step 4: continue normal computation;
        if any secondary checkpoint request is received
            discard it and continue normal execution;

```

Any Process P_j $j! = i$ and $1 \leq j \leq n$

```

if  $P_j$  receives a primary request from  $P_i$ 
    /* on receiving a primary checkpoint request */
    take a checkpoint;
    if  $DV_j[] = \text{null}$ ;
        increment  $csn_j$ ;
        continue computation;
    else
        send secondary request to each  $P_k$  such that  $DV_j[k]=1$ ;
        increment  $csn_j$ ;
        continue computation;
else if  $P_j$  receives a secondary request from  $P_i$ 
    /* on receiving a secondary checkpoint request */
    if  $P_j$  has already participated in the checkpointing algorithm
        /*  $csn_j$  is greater than the received checkpoint sequence number*/
        ignore the checkpoint request and continue computation;
    else
        if  $DV_j[] = \text{null}$ ;
            increment  $csn_j$ ;
            continue computation;
        else
            send secondary request to each  $P_k$  such that  $DV_j[k]=1$ ;
            increment  $csn_j$ ;
            continue computation;
else if  $P_j$  receives a piggy backed application message
    /* on receiving a piggy backed application message */
    if  $P_j$  has already participated in the checkpointing algorithm
        /*  $csn_j$  is greater than the received checkpoint sequence number*/
        process the message and continue computation;
    else
        take a checkpoint;
        if  $DV_j[] = \text{null}$ ;
            increment  $csn_j$ ;

```

```

        process the message;
        continue computation;
else
    send secondary request to each  $P_k$  such that  $DV_j[k]=1$ ;
    increment  $csn_j$ ;
    process the message;
    continue computation;
```

Theorem 1: Algorithm Non-Blocking produces consistent checkpoints.

Proof: In the first two steps of the algorithm for initiator process, the initiator process P_1 identifies all the application messages received from different processes that might become orphan if it takes a checkpoint by looking at its dependency vector. The initiator then sends checkpoint requests to all those processes that have sent at least one message to it asking them to take their respective checkpoints. Hence the application messages received by P_1 cannot be orphan.

Consider the pseudo code for any process P_j . In the first else if block process P_j makes sure that all the processes that are dependent on itself also take checkpoints so that there are no orphan messages that it has received. In the second else if block of the pseudo code, process P_j first takes its checkpoint, then processes the received piggybacked application message. Hence such message cannot be an orphan.

Besides, according to Observation 1 and Claim 1, no process takes more than one checkpoint, if at all, in the same execution of the checkpointing algorithm. Hence, all the checkpoints taken during the execution of the algorithm are mutually consistent. ■

3.3 Advantages

Here we state some of the main advantages that our algorithm offers over other related works.

1. Our algorithm follows a single step approach when compared to the five step approach of the algorithm reported in [4].
2. Our algorithm does not take any temporary checkpoints, and hence the overhead of converting temporary checkpoint to permanent checkpoint is eliminated, unlike in [4].
3. Our algorithm does not use mutable checkpoints as in [3]. Hence the overhead of converting them to permanent ones is eliminated
4. The number of interrupts to processes is less than those in [3], and [4].

4. Conclusion

In this paper, we have presented a single step coordinated checkpointing approach. The main features of the algorithm are:

- (1) it is non blocking and free from the avalanche effect;
- (2) it does not take any temporary or mutable checkpoint unlike in some other important related works [3], [4]. Hence, the need for converting such checkpoints to permanent ones is eliminated. It makes the algorithm faster.
- (3) absence of temporary or mutable checkpoints means that a participating process faces less number of interrupts compared to those in [3], and [4]. This also contributes to the execution speed.

5. References

- [1] K.M. Chandy, and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems", *ACM Trans. Computer Systems*, Vol 3, No. 1, pp. 63-75, Feb. 1985.
- [2] Guohong Cao, and Singhal M, "On coordinated checkpointing in distributed systems", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 9, Issue 12, pp. 1213 – 1225, Dec. 1998.
- [3] Guohong Cao, and Singhal M, "Mutable checkpoints: a new checkpointing approach for mobile computing systems", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 12, Issue 2, pp. 157-172, Feb 2001.
- [4] Kumar P, Kumar L, Chauhan R.K., and Gupta V.K., "A non-intrusive minimum process synchronous checkpointing protocol for mobile distributed systems", ICPWC 2005, IEEE International Conference on Personal Wireless Communications, pp 491-495, Jan 2005.

- [5] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel, "The Performance of Consistent Checkpointing", Proc. 11th Symp. on Reliable Distributed Systems, pp. 86-95, Oct. 1992.
- [6] L. M. Silva and J.G. Silva, "Global Checkpointing for Distributed Programs", Proc. 11th Symp. on Reliable Distributed Systems, pp. 155 –162, Oct. 1992.
- [7] R. Koo and S. Toueg, "Checkpointing and Rollback-Recovery for Distributed Systems", *IEEE Transactions on Software Engineering*, SE-13, (1), pp. 23-31, 1987.
- [8] B. Gupta, S. Rahimi, and Z. Liu, "A New Non-Blocking Synchronous Checkpointing Scheme for Distributed Systems", Proc. ISCA 20th Int. Conf. Computers and Their Applications, pp. 26 – 31, March, 2005.