A New High Performance Checkpointing Approach for Mobile Computing Systems

Bidyut Gupta†, Shahram Rahimi,† and Ziping Liu††,

†Southern Illinois University Carbondale, IL, USA, ††Southeast Missouri State University, MI, USA

Summary
In this paper, we present a single phase non-blocking coordinated checkpointing algorithm suitable for mobile computing environments. The distinct advantages that make the proposed algorithm suitable for distributed mobile computing systems are the following. It produces a consistent set of checkpoints, without the overhead of taking temporary checkpoints; the algorithm makes sure that only minimum number of processes are required to take checkpoints in any execution of the checkpointing algorithm; it uses very few control messages and the participating processes are interrupted less number of times. Performance analysis shows that our proposed approach outperforms some existing important related works.

Keywords:
Coordinated Checkpointing, Non-blocking approach, Mobile Computing Systems

Introduction

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

Checkpointing algorithms are classified into two main categories: (a) coordinated and (b) uncoordinated. In uncoordinated check pointing approach each process takes its checkpoint independently without the knowledge of the other processes. While the checkpointing approach is simple, yet it may suffer from the domino effect during recovery. In case of a failure; after recovery a CGS is found from the existing checkpoints and the system restarts from there.

In coordinated checkpointing approach, 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 check pointing algorithms are of two types: (a) blocking [7] and (b) non-blocking [3], [5], and [8]. Blocking algorithms force all relevant processes in the system to block their computation during check pointing latency and hence degrade system performance from the viewpoint of larger execution time of application programs. In non-blocking algorithms application processes are not blocked when checkpoints are being taken.

2. Related Works and Problem Formulation

The work presented in this paper shows substantial improvement in performance over the works reported in [3], [4], and [8]. Therefore, we first give a brief and clear idea about how those algorithms work and what are their complexities in terms of the number of control messages they generate.

2.1 Related works

The research in the area of coordinated checkpointing concentrates mostly on non-blocking approaches. In [3] and [4] the authors have proposed non-blocking coordinated checkpointing algorithms that require only a minimum number of processes to take checkpoints at any instant of time. In [8], it is a non-blocking coordinated check pointing algorithm; however minimality is not guaranteed. Let us now briefly state the working principles of these approaches and also the number of control messages these algorithms generate. In the following discussion, by 'number of phases of an algorithm' we mean the number of times an initiator process interacts with the other processes during the execution of the algorithm.

In [3] the authors have introduced the concept of mutable checkpoints. They are neither temporary

nor permanent checkpoints. The basic idea about the algorithm is as follows: in the first phase an initiator process first checks its own dependency vector from which it finds the processes which have sent at least one computational message to the initiator. It then sends check pointing request messages to all such dependent processes, i.e. the processes from each of which the initiator process has received at least one message. These dependent processes after taking tentative checkpoints in turn send checkpoint requests to processes that are dependent on them. This goes on until there is no further dependency found. Suppose that the minimum number of processes which need to take checkpoints is n_{min}. In the second phase processes which have received checkpoint requests send reply to the initiator. The number of such reply messages is n_{min}. In the third phase, the initiator process sends the commit message to the processes asking them to convert their tentative checkpoints to permanent ones. Here the initiator selects between the minimum cost of broadcasting the control message to n processes and the cost of sending the messages to n_{min} processes. The total number of control messages in [3] is $2*n_{min} + \min (n_{min}, n_{broad})$ where n_{broad} is the number of control messages required to broadcast to all the processes in the system. This calculation is not exact as it does not consider more complicated situations. For example, if any process which receives a check pointing request and sends a computational message to another process after taking a checkpoint, the process receiving the computational message takes a mutable checkpoint first and computes the message. This mutable checkpoint is later converted to a permanent checkpoint if it receives a check pointing request; otherwise it becomes a useless checkpoint.

In [4] the authors have proposed a five phase algorithm. It works as follows. Let us consider a distributed system with (n + 1) processes, where in the first phase an initiator process broadcasts the dependency vector request. Assume that the number of such control messages is n_{broad}. In the second phase, the initiator receives the dependency vectors from all the processes. It needs n control messages. Then the initiator creates the minimum set of processes which need to take checkpoints. Assume that the number of such processes is n_{min}. Then in the third phase the initiator takes its own tentative checkpoint and sends checkpoint request to the processes in the minimum set. It needs n_{min} control messages. Processes receiving these checkpoint requests take tentative (temporary) checkpoints. In the fourth phase, the initiator receives responses to the checkpoint request from all the processes in the minimum set. In the fifth phase, the initiator process sends the commit or abort message to all processes. It needs n_{broad} messages. Hence the total number of control messages in [4] is $n + 2*n_{min} + 2* n_{broad}$ messages. All processes that did not send or receive messages will not participate in the check pointing algorithm. Hence at any instant of time only minimum number of processes takes checkpoints.

In [8] the authors have proposed a non-blocking coordinated checkpointing algorithm where in the first phase an initiator sends check pointing request to all other processes in a system of n + 1 processes. It needs n_{broad} messages. In the second phase, dependent processes take tentative checkpoints; however all n processes send their responses to the initiator regarding if they have taken tentative checkpoints or not. This needs n control messages and in the third phase, the initiator sends the commit message to all other processes if it gets replies from all the processes within a specified time interval and takes its own checkpoint; otherwise, it sends an abort message which requires n_{broad} messages. Hence the total number of control messages in [8] is $n + 2* n_{broad}$. This work does not guarantee that only minimum number of processes will take checkpoints.

2.2 Problem formulation

In the non-blocking checkpointing algorithms discussed above all processes are supposed to take first temporary checkpoints when they receive checkpoint request and then these checkpoints are converted to permanent checkpoints. However, if a process is busy with some high priority procedure when a checkpointing request arrives at it, then it will not take a checkpoint. In such a situation, every process that has already taken a temporary checkpoint must discard it, and continue normal execution. Later the checkpointing algorithm has to be restarted again. It thus wastes time and delays further the execution of the application program. The same problem may also arise when a process receives a request to convert its temporary checkpoint to a permanent one while it is busy executing a high priority procedure. In this case also the checkpointing algorithm has to be abruptly terminated and it will restart later. Thus, such situations definitely affect the execution time of the application programs adversely.

The objective of the present work is to design a check pointing algorithm that is suitable for mobile computing environment. Mobile computing environment demands efficient use of the limited wireless bandwidth and the limited resources of mobile machines, such as battery power, memory etc. Observe that taking a temporary or a tentative checkpoint and later converting it to a permanent one needs more control messages. As a result it affects the bandwidth utilization negatively as well as it results in large number of interrupts to the mobile hosts; thereby wasting the mobile hosts' limited battery power. Therefore in the present work we emphasize on eliminating the overhead of taking temporary (tentative) checkpoints. To summarize, we have proposed a non-blocking coordinated checkpointing algorithm in which processes take permanent checkpoints directly without taking temporary checkpoints and whenever a process is busy, the process takes a checkpoint after completing the current procedure. As will be shown later that the proposed algorithm in this paper requires much fewer control messages and hence, fewer number of interrupts to each participating process compared to the other coordinated checkpointing works [3], [4], and [8]. Besides as in [3] and [4], our proposed algorithm requires only minimum number of processes to take checkpoints. It makes our algorithm suitable for mobile computing environments.

This paper is organized as follows. In Section 3 we state the system model considered in this work. We also state the date structures needed and give an example that shows how our idea works. It also contains some relevant observations. In Section 4 we have stated the algorithm. Section 5 contains a detailed performance comparison with some noted related works. In Section 6, we have discussed the suitability of our proposed algorithm in the mobile computing environment. Finally Section 7 draws the conclusion.

3. System Model and Data Structures

3,1 System model

The distributed system has the following characteristics. Processes do not share memory and they communicate via messages sent through channels. Channels can lose messages. However, they are made virtually lossless and order of the messages is preserved by some end-to-end transmission protocol. 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 may be restarted if there are further failures.

3.2 Data structures

Consider a set of n processes, $\{P_1, P_2, ..., P_n\}$ 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 an entry $DV_i[j]$ is set to 1 when P_i receives since its last checkpoint at least one message from P_j. It is reset to 0 again when process P_i takes a checkpoint. Each process P_i maintains a checkpoint sequence number csn_i. This csn_i actually represents the current check pointing 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 only one process can initiate the check pointing algorithm. This process is known as the initiator process.

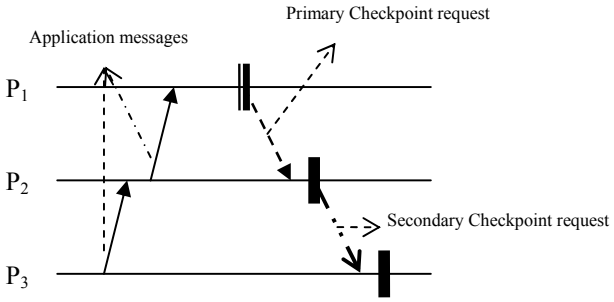

Fig. 1 Control message exchanges in our approach

We define that a process P_k is dependent on another process P_r, if process P_r since its last checkpoint has received at least one application message from process P_k. In our proposed algorithm we assume primary and secondary checkpoint request exchanges between the initiator process and the rest 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 its dependent processes asking them to take their respective checkpoints. A secondary checkpoint request denoted by R_{si} is sent from a process P_m to a process P_n which is dependent on P_m to take a checkpoint. R_{si} 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. To start with, 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 taking its checkpoint P_2 sends a secondary checkpoint request to P_3 as P_3 is dependent on P_2, Process P_3 then takes its checkpoint.

In this work, 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 have assumed that the events of taking a checkpoint and sending a checkpoint request are done atomically. Also, each process P_i piggybacks its current checkpoint sequence number with only every first outgoing application message to another process after taking

We now state the situations 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. the first time it receives a secondary checkpoint request and prior to that it has not received any primary checkpoint request or any piggybacked application message.

4. the first time it receives an application message piggybacked with the checkpoint sequence number, and prior to that it has not received any primary or secondary checkpoint request message.

Before the formal statement of the algorithm, its working principle is illustrated with an example. For a clear understanding, the above four different possible scenarios are discussed.

3.3 An Illustration

The behavior of each process in our approach is explained with the help of the following example. Unless otherwise mentioned a checkpoint request represents either a primary request or a secondary request. Note that an application message with piggybacked checkpoint sequence number, which may force a checkpoint to be taken at the receiving process may also be viewed as a checkpoint request. In our work a checkpoint means a permanent checkpoint.

Consider the distributed system as shown in the Fig. 2. Assume that process P_2 initiates the check pointing algorithm. First process P_2 takes its permanent 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 its checkpoint $C_{2,1}$ these messages will become orphan if P_1, P_3, and P_4 do not take checkpoints. Therefore P_2 sends primary checkpoint request R_1 ($csn_2 = 1$) to P_1, P_3, and P_4. After sending the primary checkpoint request process P_2 increments its checkpoint sequence number csn_2 to 2 and finishes its participation associated with the current execution of the algorithm and continues with its normal computation. It shows the non-blocking nature of our approach.

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 also 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 finds that process P_7 has sent a message to it. Hence it sends

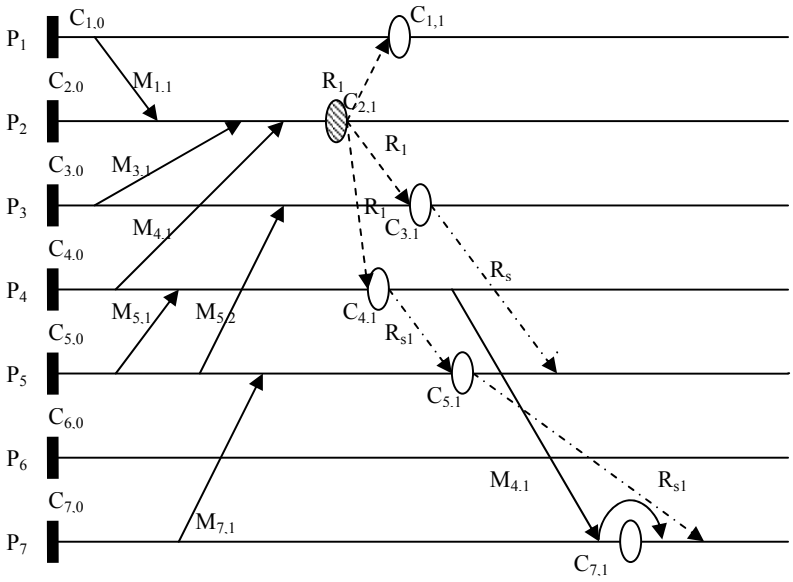

Fig. 2 An example of our checkpointing approach

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 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 its 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 its checkpoint for the current execution of the algorithm.

In the above example we observe that P_7 sent a message $M_{7,1}$ to P_5. So even if there was no such piggybacked message as $M_{4,1}$, process P_7 would eventually receive the secondary checkpointing request $R_{s,1}$ from P_5 and take its checkpoint $C_{7,1}$.

Observe that because of the non-blocking nature of the algorithm the following situation may arise as well. Consider that there was no such message as $M_{7,1}$; that is, assume that P_7 has not sent any application message to any process at all. However, assume that it receives the piggybacked message $M_{4,1}$ from P_4. In our approach P_7 will take its checkpoint and then process the message and then would behave like any other process involved in the checkpointing approach. This helps in the advancement of the consistent state of process P_7. This means that in the event of a failure occurring after P_7 takes its checkpoint $C_{7,1}$, process P_7 can restart from this checkpoint, instead of its initial state $C_{7,0}$ after the system recovers from the failure.

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, because of some typical communication pattern among the processes.

Avalanche Effect: Consider the following situation: suppose process P_i initiates the coordinated checkpointing scheme. It takes a checkpoint; checks its dependency vector and sends primary requests 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 takes a checkpoint; checks its own dependency vector and sends a secondary request to P_k. Process P_k in turn takes a checkpoint; checks its dependency vector and sends a request to P_r 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 the message communication pattern is such that process P_i has taken a checkpoint after receiving a primary checkpointing request, and then has sent a secondary checkpoint request to P_k; process P_k takes its checkpoint and after checking its dependency vector $DV_k[]$ sends a secondary request to P_r, and P_r acts similarly and sends a secondary request to P_q. Following the same way finally some process P_s sends a secondary checkpoint request to P_i because in its dependency vector, $DV_s[i] = \neq 0$, and P_i in turn again sends a secondary request to P_k and so on. Therefore, it appears that the chain of requests form a loop as P_i, P_k, P_r, P_q,…..,P_s, P_i , P_k, … for a possible avalanche effect to occur.

Consider process P_s. It has received a secondary request and taken a checkpoint, and then has sent a secondary checkpoint request to P_i since $DV_s[i]$ is not zero. Let us examine if this scenario leads to a possible avalanche effect.

Since $DV_s[i]$ is not zero, therefore P_s has received at least one application message, say M_i from process P_i before P_i takes its checkpoint. Observe that process P_i must have sent this message (s) to P_s before it has initiated the checkpointing process; otherwise this message would have been a piggybacked one and then according to our approach P_s would have taken a checkpoint first when it received the piggybacked message and then process it. Obviously then P_s would just ignore the secondary request it has received, because by then it has finished its participation in the checkpointing process and as a result it would not send any secondary checkpointing request to P_i. However, this is not the case with Ps, because it has sent a secondary request to P_i. As pointed out above, since P_i sent this message (s) before taking its checkpoint, therefore this message can not be an orphan and as a result P_i does not need to take any checkpoint when it receives the secondary request from P_s. Therefore P_i will just ignore the request. Hence there can not exist a chain of requests forming the loop of processes as P_i, P_k, P_r, P_q,…..,P_s, P_i , P_k, … which might otherwise lead to a possible avalanche effect. Hence the assumption that avalanche effect is possible is not valid in our approach. ■

In the following algorithm we have considered all four situations mentioned in Section 3.2 about when a process takes a checkpoint.

4. Algorithm Non_Blocking

As in any conventional coordinated check pointing scheme at any instant of time any one process can initiate the check pointing algorithm.

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 $1 <= k <= n$
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 $1 <= j <= n$

if P_j receives a primary checkpoint request from P_i
take a checkpoint; */* if P_j is busy with other high priority job, it takes a checkpoint after the job ends; otherwise it takes a checkpoint immediately */*
if $DV_j[] = null$;
increment csn_j;
continue computation;
else
send secondary checkpoint request to each P_k such that $DV_j[k] = 1$;
increment csn_j;
continue computation;

else if P_j receives 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
take a checkpoint; */* if P_j is busy with other high priority job, it takes a checkpoint after the job ends; otherwise it takes a checkpoint immediately */*
if $DV_j [] = null$;
increment csn_j;
continue computation;
else
send secondary checkpoint 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
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 P_j is busy with other high priority job, it takes a checkpoint after the job ends; otherwise it takes a checkpoint immediately*/*
if $DV_j [] = null$;
increment csn_j;
process the message;
continue computation;
else
send secondary checkpoint 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 a consistent global state of the system.

Proof: In the first two steps of the algorithm for initiator process, the initiator process P_i 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 primary checkpoint requests to all those processes that have sent at least one message to it asking them to take their respective checkpoints. Hence any application message received by P_i cannot be an orphan.

Consider the pseudo code for any process P_j. Process P_j makes sure that all processes from which it has received messages 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 if needed, then processes the received piggybacked application message. Hence such a message cannot be an orphan. Hence the algorithm generates a consistent global state (CGS) of the system. ■

Claim 2: Number of processes taking checkpoints is minimum.

Proof: According to Observation 1 a process takes a checkpoint if and only if it is the initiator, or it receives either a primary checkpoint request or a secondary checkpoint request or a piggybacked application message. This means that any process that is not an initiator or that does not receive any of the above mentioned control messages, does not take a checkpoint. Hence the proof follows. ■

Summery of the main advantages our algorithm are as follows:

1. Our algorithm follows a one phase approach when compared to the three phase and five phase approaches in [3], [8] and [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] and [8].
3. Our algorithm does not use mutable checkpoints as in [3]. Hence the overhead of converting them to permanent ones is eliminated. Also our work does not allow any process to take useless checkpoints.
4. The number of interrupts to processes is less than those in the algorithms [3], [4], and [8].

A detailed estimate of the number of control messages (hence, interrupts to the processes) needed by our algorithm and the related works is given in the next section.

5. Performance

The main advantage of our algorithm over the algorithms [3], [4], and [8] is that the cost for determining a consistent state of the system is much less compared to the ones in [3], [4], and [8]. We have presented the comparison of performance of the above three algorithms with our algorithm in Table 1. .

For ease of interpretation of the performance parameters we consider an n+1 process distributed system. Let n_{min} represent the minimum number of processes that need to take a checkpoint, C_{air} be the cost of sending a message from one process to another, and n_{broad} be the cost of broadcasting a message to all processes in the system.

The cost to complete the checkpoint process using algorithm [3] is given as $2*n_{min}*C_{air}$ + min ($n_{min}*C_{air}$, n_{broad}) in the best case. As mentioned earlier, in algorithm [3] first the initiator sends control messages to minimum number of processes that need to take a checkpoint each. The cost for this is $n_{min}*C_{air}$. When each process takes a tentative checkpoint it replies back to the initiator acknowledging the request to take a checkpoint. Hence a cost of $2*n_{min}*C_{air}$ is needed. When the initiator receives the acknowledgement from all the processes, it informs them to convert their respective tentative checkpoints into permanent checkpoints which contributes further a cost of $n_{min}*C_{air}$. If the cost of broadcasting the message is less than sending the messages to n_{min} processes then the message can be broadcasted. In this way, the algorithm in [3] generates a consistent set of checkpoints. The total cost for such a generation is $2*n_{min}*C_{air}$ + min ($n_{min}*C_{air}$, n_{broad}) in the best case (secondary and tertiary dependencies are not considered).

In [4] the initiator broadcasts dependency vector request to all the n processes the cost of which is n_{broad}. The initiator receives the vectors from the n processes the cost of which is $n*C_{air}$. Initiator calculates the minimum dependency set from the dependency vectors and sends checkpoint request message to the minimum number of processes that need to take checkpoints, the cost of which is

Table 1: Performance Comparison of the Checkpointing Algorithms.

	Mutable [3]	Non intrusive [4]	CCUML [8]	Our algorithm
Cost (best case)	$2*n_{min}*C_{air} + \min(n_{min}*C_{air}, n_{broad})$	$n*C_{air} + 2 * n_{min}*C_{air} + 2 * n_{broad}$	$n*C_{air} + 2*n_{broad}$	$n_{min}*C_{air}$
Useless checkpoints	present	nil	nil	nil
Temporary checkpoints	present	present	present	no
Non-Blocking	Yes	Yes	Yes	Yes
Number of checkpoints	n_{min}	n_{min}	n+1	n_{min}

$n_{min}*C_{air}$. These processes reply to the initiator after taking temporary checkpoints, the cost of which is $n_{min}*C_{air}$. Finally the initiator broadcasts a commit message to all the processes, the cost of which is n_{broad}. In this way the algorithm in [4] generates a consistent set of checkpoints. The cost for such a generation is $n*C_{air} + 2 * n_{min}*C_{air} + 2 * n_{broad}$.

In [8] the initiator broadcasts the checkpoint request to all processes the cost of which is n_{broad}. The initiator receives replies from the n processes the cost of which is $n*C_{air}$. Finally the initiator broadcasts a commit message to all processes to convert their temporary checkpoints to permanent ones, the cost of which is which is n_{broad}. Hence the total cost in [8] is $n*C_{air} + 2*n_{broad}$. Note that it does not guarantee that only minimum number of processes will take checkpoints.

In Fig. 3 and Fig. 4 the ordinate represents the cost of sending control messages to complete the check pointing algorithm in the best case for the four algorithms. In Fig. 3 we assume $n_{broad} = n *C_{air}$; that is cost of broadcasting a message is equal to the cost of sending n messages.

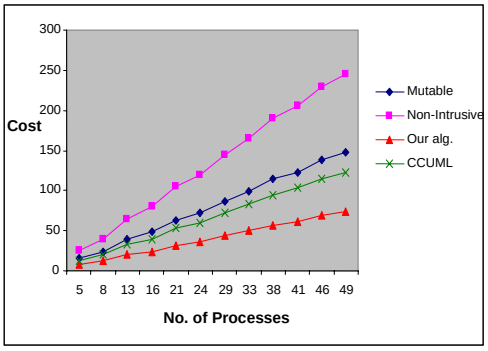

Fig. 3 Comparison of costs when $n_{broad} = n * C_{air}$

In Fig. 4 we assume the cost of broadcasting is equal to the cost of sending a single message; that is $n_{broad} = C_{air}$. Fig. 3 clearly demonstrates the better performance of our approach than the ones in [3], [4], and [8]. In Fig. 4, the cost for sending control messages in our approach and the one in [8] are same. However, the work in [8] does not offer minimum number processes to take checkpoints.

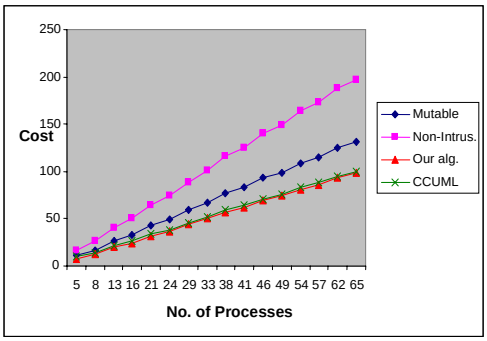

Fig. 4 Comparison of costs when $n_{broad} = C_{air}$

6. Suitability for Mobile Computing Environment

Consider a distributed mobile computing environment. In such an environment, only limited wireless bandwidth is available for communication among the computing processes. Besides, the mobile hosts have limited battery power and limited memory. Therefore, it is required that, any distributed application *P* running in such an environment must make efficient use of the limited wireless bandwidth, and mobile hosts' limited battery power and memory. Below we justify why the proposed algorithm will be more effective than the ones in [3], [4], [8], and [10] in mobile computing environment.

The following advantages make our approach more suitable for the mobile computing environment:

(1) Our algorithm is a single phase algorithm which clearly indicates that it terminates fast which is an important advantage when considering limited battery power of mobile hosts. None of the other related works [3], [4], [8], [10] is a single phase algorithm.

(2) As is seen from Table 1, the presented algorithm uses the minimum number of control messages. It definitely offers much better bandwidth utilization than the above mentioned related works.

(3) Minimum number of control messages means that mobile hosts face minimum number of interrupts compared to the other works. It saves the limited battery power of the mobile machines significantly.

(4) In our algorithm, processes neither take any useless checkpoints unlike in [3], nor they take any unnecessary local checkpoints unlike in [10]. This offers better utilization of the mobile hosts' limited memory.

7. Conclusion

In this paper, we have presented a single phase non-blocking coordinated checkpointing approach suitable for mobile computing environment. The main features of the algorithm are: (1) it is free from the avalanche effect and minimum number of processes take checkpoints; (2) it does not take any temporary, tentative, or mutable checkpoint unlike in some other important related works [3], [4], [8]. Absence of temporary, tentative, or mutable checkpoints (hence some possible useless checkpoints) means that much fewer number of control messages are needed. These advantages make the proposed algorithm more suitable for mobile computing environment than the algorithms mentioned above.

References

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

[2]. G. Cao, and M. Singhal, "On coordinated checkpointing in distributed systems," Parallel and Distributed Systems, IEEE Transactions on Parallel and Distributed Systems, vol. 9, Issue 12, pp. 1213 – 1225, Dec. 1998.

[3] G. Cao, and M. Singhal, "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] P. Kumar, L. Kumar, R.K. Chauhan, and V.K. Gupta, "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, New Delhi.

[5] E.N.Elnozahy, D.B.Johnson, and W. Zwaenepoel, "The Performance of Consistent Checkpointing," Proceedings of 11th Symp. On Reliable Distributed Systems, pp. 86-95, October 1992, Houston.

[6] L. M. Silva and J.G. Silva, "Global Checkpointing for Distributed Programs," Proceedings of 11th Symp. On Reliable Distributed Systems, pp. 155 –162, October 1992, Houston.

[7] R. Koo and S. Toueg, "Checkpointing and Rollback-Recovery for Distributed Systems", IEEE Transactions on Software Engineering, SE-13, (1), pp. 23-31, January 1987.

[8] S. Neogy, A. Sinha, P.K. Das, "CCUML: a check pointing protocol for distributed system processes," TENCON 2004. 2004 IEEE Region 10 Conference vol. B, no.2, pp. 553 – 556, November 2004, Thailand.

[9] 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, New Orleans.

[10] R. Prakash, and M.Singhal, "Low-Cost Check pointing and Failure Recovery in Mobile Computing Systems," IEEE Transactions on Parallel and Distributed Systems, vol. 7, no. 10, pp. 1035-1048, October 1996,

[11] D. Manivannan, and M. Singhal, "Asynchronous Recovery Without Using Vector Timestamps," Journal of Parallel and Distributed Computing, vol. 62, no. 12, pp. 1695-1728, December 2002.

Bidyut Gupta received his PhD in Computer Science and his MTech degree in Electronics Engineering from the University of Calcutta, India. Currently, he is a professor of computer science and the graduate director for Computer Science department at the Southern Illinois University Carbondale.

Shahram Rahimi received his PhD in Scientific Computing and his MS degree in Computer Science from the University of Southern Mississippi in 1998 and 2002 respectively, and his BS from National University of Iran (Tehran) in 1992. Currently, he is an assistant professor at Southern Illinois University and the Editor-in-Chief of the International Journal of Computational Intelligence Theory and Practice.

Ziping Liu is an assistant professor at the Computer Science Department, South East Missouri State University.