ISSN (Online): 1694-0784
ISSN (Print): 1694-0814

A Minimum-Process Coordinated Checkpointing Protocol For Mobile Distributed System

Praveen Kumar[1] and Ajay Khunteta[2]

[1] **Department of Computer Science & Engineering**
Meerut Institute of Engineering & Technology, Meerut, India, Pin-125005

[2]**Singhaniya University**
Pechri, Rajasthan, India

Abstract

While dealing with Mobile Distributed systems, we come across some issues like: mobility, low bandwidth of wireless channels and lack of stable storage on mobile nodes, disconnections, limited battery power and high failure rate of mobile nodes. These issues make traditional checkpointing techniques designed for Distributed systems unsuitable for Mobile environments. In this paper, we design a minimum process algorithm for Mobile Distributed systems, where no useless checkpoints are taken and an effort has been made to optimize the blocking of processes. We propose to delay the processing of selective messages at the receiver end only during the checkpointing period. A Process is allowed to perform its normal computations and send messages during its blocking period. In this way, we try to keep blocking of processes to bare minimum. We captured the transitive dependencies during the normal execution by piggybacking dependency vectors onto computational messages. In this way, we try to reduce the Checkpointing time by avoiding formation of Checkpointing tree. The Z-dependencies are well taken care of. The proposed scheme forces zero useless checkpoints at the cost of very small blocking.

1. Introduction

Checkpoint is defined as a designated place in a program at which normal process is interrupted specifically to preserve the status information necessary to allow resumption of processing at a later time. A checkpoint is a local state of a process saved on stable storage. By periodically invoking the checkpointing process, one can save the status of a program at regular intervals [3], [4]. If there is a failure, one may restart computation from the last checkpoints, thereby, avoiding repeating computation from the beginning. The process of resuming computation by rolling back to a saved state is called rollback recovery [6]. In a distributed system, since the processes in the system do not share memory, a global state of the system is defined as a set of local states, one from each process. The state of channels corresponding to a global state is the set of messages sent but not yet received [7].

A message whose receive event is recorded, but its send event is lost. A global state is said to be "consistent" if it contains no orphan message. To recover from a failure, the system restarts its execution from a previous consistent global state saved on the stable storage during fault-free execution. In distributed systems, checkpointing can be independent, coordinated [3], [8], [11], [15] or quasi-synchronous [2], [9]. Message Logging is also used for fault tolerance in distributed systems [7], [14]. Under the asynchronous approach, checkpoints at each process are taken independently without any synchronization among the processes. Because of absence of synchronization, there is no guarantee that a set of local checkpoints taken will be a consistent set of checkpoints. It may require cascaded rollbacks that may lead to the initial state due to domino-effect [7].

In coordinated or synchronous Checkpointing, processes take checkpoints in such a manner that the resulting global state is consistent. Mostly it follows two-phase commit structure [3], [8], [11], [22]. In the first phase, processes take tentative checkpoints and in the second phase, these are made permanent. The main advantage is that only one permanent checkpoint and at most one tentative checkpoint is required to be stored. In the case of a fault, processes rollback to the last checkpointed state.

It avoids the domino-effect without requiring all checkpoints to be coordinated [2], [7], [9]. In these protocols, processes take two kinds of checkpoints, local and forced. Local checkpoints can be taken independently, while forced checkpoints are taken to guarantee the eventual progress of the recovery line and to minimize useless checkpoints. P_j is directly dependent upon P_k only if there exists *m* such that P_j receives *m* from P_k in the current CI and P_k has not taken its permanent checkpoint after sending *m*. A process P_i is in the minimum set only if checkpoint initiator process is transitively dependent upon it. In minimum-process coordinated checkpointing algorithms, only a subset of interacting processes (called minimum set) are required to take checkpoints in an initiation.

The Chandy-Lamport [6] algorithm is the earliest non-blocking all-process coordinated checkpointing algorithm. In this algorithm, *markers* are sent along all channels in the network which leads to a message complexity of $O(N^2)$, and requires channels to be FIFO. Elnozahy et al. [8] proposed an all-process non-blocking synchronous checkpointing algorithm with a message complexity of O(N). In coordinated checkpointing protocols, we may require piggybacking of integer csn (checkpoint sequence number) on normal messages [5], [8], [13], [19], [22].

The existence of mobile nodes in a distributed system introduces new issues that need proper handling while designing a checkpointing algorithm for such systems. These issues are mobility, disconnection, finite power source, vulnerable to physical damage, lack of stable storage etc. These issues make traditional checkpointing techniques unsuitable to checkpoint mobile distributed systems [1], [5], [15]. To take a checkpoint, an MH has to transfer a large amount of checkpoint data to its local MSS over the wireless network. Since the wireless network has low bandwidth and MHs have low computation power, all-process checkpointing will waste the scarce resources of the mobile system on every checkpoint. Prakash and Singhal [15] gave minimum-process coordinated checkpointing protocol for mobile distributed systems.

A good checkpointing protocol for mobile distributed systems should have low overheads on MHs and wireless channels and should avoid awakening of MHs in doze mode operation. The disconnection of MHs should not lead to infinite wait state. The algorithm should be non-intrusive and should force minimum number of processes to take their local checkpoints [15]. In minimum-process coordinated checkpointing algorithms, some blocking of the processes takes place [4], [11], or some useless checkpoints are taken [5], [13], [19].

Cao and Singhal [5] achieved non-intrusiveness in the minimum-process algorithm by introducing the concept of mutable checkpoints. The number of useless checkpoints in [5] may be exceedingly high in some situations [19]. Kumar et. al [19] and Kumar et. al [13] reduced the height of the checkpointing tree and the number of useless checkpoints by keeping non-intrusiveness intact, at the extra cost of maintaining and collecting dependency vectors, computing the minimum set and broadcasting the same on the static network along with the checkpoint request.

Koo and Toeg [11], and Cao and Singhal [4] proposed minimum-process blocking coordinated checkpointing algorithms. Neves et al. [12] gave a loosely synchronized coordinated protocol that removes the overhead of synchronization. Higaki and Takizawa [10] proposed a hybrid checkpointing protocol where the mobile stations take checkpoints asynchronously and fixed ones synchronously. Kumar and Kumar [29] proposed a minimum-process coordinated checkpointing algorithm where the number of useless checkpoints and blocking are reduced by using a probabilistic approach. A process takes its mutable checkpoint only if the probability that it will get the checkpoint request in the current initiation is high. To balance the checkpointing overhead and the loss of computation on recovery, P Kumar [24] proposed a hybrid-coordinated checkpointing protocol for mobile distributed systems, where an all-process checkpoint is taken after executing minimum-process checkpointing algorithm for a certain number of times.

Transferring the checkpoint of an MH to its local MSS may have a large overhead in terms of battery consumption and channel utilization. To reduce such an overhead, an incremental checkpointing technique could be used [16]. Only the information, which changed since last checkpoint, is transferred to the MSS.

In the present study, we purpose a minimum process coordinated checkpointing algorithm for Mobile Distributed Systems in which no useless checkpoints are taken and the blocking of processes is reduced to bare minimum.

2. System Model

We use the system model presented in [2], [4]. In this model, a mobile computing system consists of n mobile hosts (MHs), and m mobile support stations (MSSs), where n > m. A cell is a logical or geographical coverage area under an MSS. An MH can directly communicate with an MSS M_i only if it is present in the cell serviced by Mi. At any time, an MH belongs to only one cell or may be disconnected. The static network provides reliable First-In-First-Out (FIFO) delivery of messages between any two MSSs with arbitrary message latency. Similarly, the wireless network within a cell ensures reliable FIFO delivery of messages between an MSS and an MH.

In this paper, we consider a distributed computation in a mobile computing system that consists of N processes, running concurrently on different MHs or MSSs. For simplicity, we assume that each MH runs one process. Message passing is the only way of communication. The computation is asynchronous. The processes do not share memory or clock. Each process progresses at its own speed and messages are exchanged through reliable channels, whose transmission delays are finite but arbitrary. A process in the cell of MSS means the process is either running on the MSS or on an MH supported by it. It also includes the processes of MHs, which have been disconnected from the MSS but their checkpoint related information is still with this MSS. We also assume that the processes are non-deterministic. The i^{th} CI (checkpointing interval) of a process denotes all the computation performed between its ith and $(i+1)^{th}$ checkpoint, including the ith checkpoint but not the $(i+1)^{th}$ checkpoint.

3. Basic Idea

During the execution of checkpointing algorithm, a process P_i may receive m from P_j such that P_j has taken its tentative checkpoint for the current initiation whereas P_i has not taken. If P_i processes m and it receives checkpoint request later on and takes its checkpoint, then m will become orphan in the recorded global state. We propose that such messages should be buffered at the receiver end. In the present discussion, P_i processes m only after taking its tentative checkpoint if it is a member of the minimum set; otherwise, P_i processes m after getting the exact minimum set and knowing that it is not a member of the minimum set.

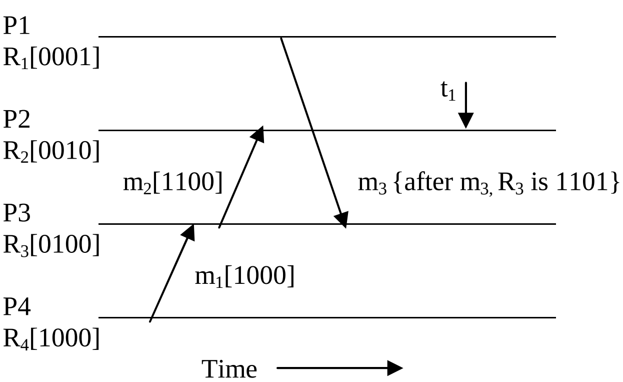

Fig. 3.1 Basic Idea

In the figure 3.1 P_4 sends m_1 to P_3 along with its own dependency vector R_4[1000]. When P_3 receives m_1 it updates its own dependency vector by taking logical OR of R_4 & R_3[0100], which comes out to be 1100. When P_3 send m_2 to P_2, it appends R_3[1100] along with m_2. When P_2 receive m_2, it updates its own dependency vector R_2 by taking logical OR of R_2 and R_3, which comes out to be [1110]. In this way, partial transitive dependencies are captured during normal computation. It should be noted that all the transitive dependencies are not captured during normal computation. At time t1, the dependency vector of P_2 shows that P_2 is not transitively dependent upon P_1, due to m_3 and m_2.

3.1 Example

We explain our algorithm with an example. P_1, P_2, P_3, P_4 and P_5 are processes with initial dependency set [00001], [00010], [00100], [01000] and [10000], respectively.

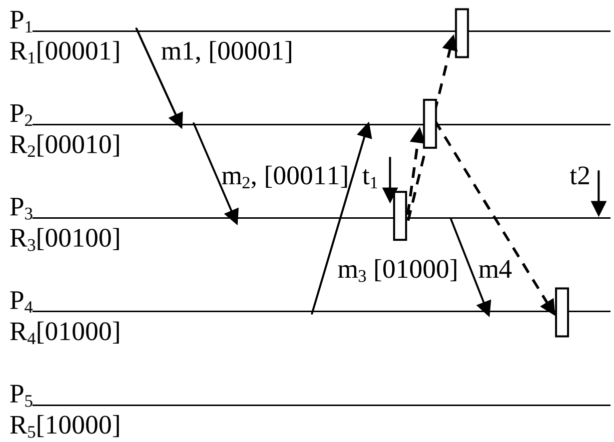

Fig 3.2 An Example

{⟶ indicate message, - - -► indicate request of checkpoint, R_i represent the set of dependency.}

At time t_1, P_3 initiates checkpointing with dependency set [00111], therefore it sends the checkpointing request to P_1 and P_2 only, which in turn takes their tentative checkpoints. After taking its tentative checkpointing, P_3 sends m_4 to P_4. When P_4 receives m_4, its find that P_3 has taken its tentative checkpoint before sending m_4 because CSN (checkpoint sequence number) of P_3 is 1 at time of sending m_4; therefore, P_4 buffers m_4. When P_2 takes its tentative checkpoint, it find that it is dependent upon P_4 due to m_3 and P_4 is not in the minimum set of dependency computed so far; therefore, P_2 send checkpoint request to P_4. After taking its tentative checkpoint, P_4 process m_4. At time t_2, P_3 receives response from all processes and sends commit request to all processes along with exact minimal set of dependency, which is not shown in the figure. Hence, the messages, which can become orphan, are buffered at the receiver end. A process processes the buffered messages only after taking its tentative checkpoint or after getting the commit request.

4. Data Structures

Here, we describe the data structures used in the proposed checkpointing protocol. A process on MH that initiates checkpointing, is called initiator process and its local MSS is called initiator MSS. If the initiator process is on an

MSS, then the MSS is the initiator MSS. All data structures are initialized on completion of a checkpointing process, if not mentioned explicitly.

Pr_csn_i: A monotonically increasing integer checkpoint sequence number for each process. It is incremented by 1 on tentative checkpoint.

td_vect_i []: It is a bit array of length n for n process in the system. $td\_vect_i[j]$ =1 implies P_i is transitively dependent upon P_j. When P_i receives m from P_j such that P_j has not taken any permanent checkpoint after sending m then P_i sets $td\_vect_i[j]$=1. When P_i commit its checkpoint, it sets $td\_vect_i[]$ =0 for all processes except for itself which is initialized to 1.

$chkpt\text{-}st_i$: A boolean which is set to '1' when P_i takes a tentative checkpoint; on commit or abort, it is reset to zero

m_vect[]: A bit array of size n for n processes in the systems. When P_i starts checkpointing procedures, it computes tentative minimum set as follows: m_vect[j] = $td\_vect_i[j]$ where j=1, 2, …., n.

TC[]: An array of size n to save information about the processes which have taken their tentative checkpoints. When process P_j takes its tentative checkpoint then j^{th} bit of this vector is set to 1. It is initialized to all zeros in the beginning of the checkpointing process. It is maintained by the checkpoint initiator MSS only.

Max_time: it is a flag used to provide timing in checkpointing operation. It is initialized to zero when timer is set and becomes '1' when maximum allowable time for collecting global checkpoint expires.

MSS_plist[]: A bit array of length n for n processes which is maintained at each MSS $MSS\_plist_K[j]$ =1 implies each process P_j is running on MSS_k. If P_j is disconnected, then it checkpoint related information is on MSS_k.

MSS_chk_taken: A bit array of length n bits maintained by the MSS. MSS_chk_taken [j]=1 implies P_j which is in the cell of MSS has taken its tentative checkpoint.

MSS_chk_request: A bit array of length n at each MSS. The j^{th} bit of this array is set to '1' whenever initiator sends the checkpoint request to P_j and P_j is in the cell of this MSS.

MSS_fail_bit: A flag maintained on every MSS, initialized to '0'; set to '1' when any process in the cell of MSS fails to take tentative checkpoint.

P_{in}: The process which has initiated the checkpointing operation.

MSS_{in}: The MSS, which has P_{in} in its cell.

p_csn_{in}: checkpoint sequence number of initiator process.

g_chkpt: A flag which indicates that some global checkpoint is being saved.

csn[]: An array of size n, maintained on every MSS, for n processes. csn[i] represents the most recently committed checkpoint sequence number of P_i. After the commit operation, if m_vect[i] =1 then csn[i] is incremented. It should be noted that entries in this array are updated only after converting tentative checkpoints in to permanent checkpoints and not after taking tentative checkpoints.

m_vect1[]: An array of size n maintained on every MSS. It contains those new processes which are found on getting checkpoint request from initiator.

m_vect2 []: An array of size n. for all j such that m_vect1 [j] $\neq$0, m_vect2= m_vect2 $\cup$ m_vect1.

m_vect3[]: An array of length n; on receiving m_vect3[], m_vect[], m_vect1[] along with checkpoint request [c_req] or on the computation of m_vect1[] locally:
m_vect3[]=m_vect3[] ∪ c_req.m_vect3[];
m_vect3[]=m_vect3[]∪m_vect[];
m_vect3[]=m_vect3[]∪c_req.m_vect1[];
m_vect3[]=m_vect3[] ∪ m_vect1[];
m_vect3[] maintains the best local knowledge of the minimum set at an MSS.

4.1 Computation of m_vect[], m_vect1[], m_vect2[], m_vect3[]:

1. Suppose a process P_r wants to initiate checkpointing procedure. Its send its request to its local MSS, say MSS_r.. MSS_r maintains the dependency vector of P_r (say $td\_vect_r[]$). MSS_r coordinates checkpointing on behalf of P_r. It computes tentative minimum set as follows:

$$\forall_{i=1,n} \ m\_vect[i] = td\_vect_r[i]$$

2. On receiving m_vect[] from MSS_r, any MSS (say MSS_S) computes the m_vect1[] as follows:

Suppose MSSs maintains the process P_j such that P_j $\in$ MSSs and $P_j \in$ m_vect

$\forall i$, m_vect1[i]=1 iff m_vect[i]=0 and $td\_vect_j[i]$=1

m_vect1[] maintains the new processes found for the minimum set when a process receives the checkpoint request.

m_vect2=m_vect2 U m_vect1

$\forall$ i, m_vect1[i]=0

3. m_vect3= m_vect U m_vect2

MSS_{in} sends c_req to MSS_s along with m_vect[]and some process (say P_k) is found at MSS_s, which takes the checkpoint to this c_req. All MSSs maintains the processes of minimum set to the best of their knowledge in m_vect3. It is required to minimize duplicate checkpoint requests. Suppose, there exists some process (say P_l) such that P_k is directly dependent upon P_l and P_l is not in the m_vect3, then MSS_s sends c_req to P_l. The new processes found for the minimum set while executing a potential checkpoint request at an MSS are stored in m_vect1. When an MSS finds that all the local processes, which were asked to take checkpoints, have taken their checkpoints, it sends the response to the MSS_{in} along with m_vect2; so that MSS_{in} may update its knowledge about minimum set and wait for the new processes before sending commit. In this way, MSS_{in} sends commit only if all the processes in the minimum set have taken their tentative checkpoints.

5. The Checkpointing Protocol

As the wireless bandwidth is a scarce commodity in mobile systems; therefore; we impose minimum burdon on wireless channels. The local MSS of an MH acts on behalf of the process running on MH.

We piggyback checkpoint sequence numbers and dependency vectors onto normal computation messages, but this information is not sent on wireless channels. The local MSS of an MH, strips all the additional information from the computation message and sends it to the concerned MH. The dependency vector of a process running on an MH is maintained by its local MSS.

Our algorithm is distributed in nature in the sense that any process can initiate checkpointing. If two processes initiate checkpointing concurrently, then the checkpoint imitator of the lower process ID will prevail. The local MSS of a process coordinates checkpointing on its behalf. Suppose two processes P_i and P_j starts checkpointing concurrently and MSS_p and MSS_q are their local MSS respectively then MSS_p and MSS_q will send checkpoint requests along with tentative minimum set to all the MSS's. MSS_p will receive the checkpoint request of MMS_q and MMS_q will receive the checkpoint request of MSSp. Suppose Process-ID of P_i is less than Process-ID of P_j, then the checkpoint initiates of P_i will prevail. Any other MSS will automatically ignore the request of P_j because every MSS will compare the process id of P_i and P_j.

We propose that any process in the system can initiate the checkpointing operation. When a process P_{in} starts checkpointing procedure, it send its request to its local MSS say MSS_{in}. MSS_{in} computes the tentative minimum set m_vect[] as follows:

$\forall_{i=1,n}$ m_vect[i] = td_vect[i]

MSS_{in} coordinates checkpointing process on behalf of $P_{in.}$
We want to emphasize that $td\_vect_{in}[]$ contains the processes on which P_{in} transitively depends and the set is not complete.

MSS_{in} sends c-req to all MSS's along with $m\_vect_{in}[]$. When an MSS say MSS_p receives c-req; it sends the c-req to all such process which are running in it and are also the member of $m\_vect_{in}[]$. Suppose P_j gets the checkpoint request at MSS_p Now we find any process P_k such that P_k does not belong to $m\_vect_{in}[]$ and P_k belongs to $td\_vect_j[]$. In this case, P_k is also included in the minimum set. During checkpointing suppose P_i takes it tentative checkpoint and after that it send m to P_j such that P_j has not taken it tentative checkpoint at the time of receiving m. If P_j receive m and it gets checkpoint request later on then m will become orphan. In order to handle this situation, we buffer m at P_j. P_j receive m after taking its tentative checkpoint if it is member of minimum set; otherwise it process m on commit.

For a disconnected MH that is a member of minimum set, the MSS that has its disconnected checkpoint, converts its disconnected checkpoint into tentative one. When a MSS learns that its concerned processes in its cell have taken their tentative checkpoints, it sends the response to MSS_{in}. On receiving positive response from all concerned MSSs, the MSS_{in} issues the commit request to all MSSs. On commit when a process learns that it has buffered some message and has not received the formal tentative checkpointing request from any process, then it processes the buffered messages.

5.1 Formal Outline of the checkpointing Algorithm:

5.1.1 Actions taken when P_i sends m to P_j:

send(P_i, P_j, m, pr_csn_i,$td\_vect_i[]$);

//P_i piggybacks its own csn and transitive dependency vector onto m.

5.1.2 Algorithm executed at initiator MSS (say MSS_{in})

Suppose P_{in} initiates checkpointing. P_{in} sends the request to MSS_{in}. MSS_{in} computes m_vect [Refer section 4.1].

(1)On the basis of computed m_vect, MSS_{in} computes m_vect1, m_vect2, m_vect3 [Refer section 4.1].

(2) m_vect = m_vect3.

(3) MSS_{in} sends c_req to all MSSs along-with m_vect[].

(4) Set max-time.

(5) Wait for response.

(6) On receiving response (P_{in}, MSS_{in}, MSS_s, mss_ chk_taken, m_vect2, mss_fail_bit) or at max_time

(a) If (max_time)OR(mss_fail_bit){ send message abort (P_{in}, MSS_{in}, pr_csn_{in}} to all MSS_s, Exit; //Maximum allocated time expired or some process failed to take checkpoint

(b) m_vect[] = m_vect[]U m_vect2[]. ["U" is a set union operator]

(c) TC[] = TC[] U mss_chk_taken[]

(7) For (k=0;k<n; k++)

If ($\exists$ k such that TC[k] $\neq$ m_vect[k]) then go to step 5;

(8) Send message commit (P_{in}, MSS_{in},pr_csn_{in}, m_vect[]) to all MSS_s; // m_vect[] is the exact minimum set//

5.1.3 Algorithm Executed at a process P_j on receiving of m from P_i:

Case 1: If ($m.pr\_csn_i$ = = csn[i])// P_i has not taken its tentative checkpoint before sending m

{ rec(m);

td_vect$_j$[i]=1};
Case 2: If (m.pr_csn$_i$<csn[i]; rec (m)); P_i has taken some permanent checkpoint // after sending m
Case 3: If((m.pr_csn$_i$>csn[i]) AND (pr_csn$_j$>csn[j]));
{rec (m); td_vect$_j$[i]=1} //P_i & P_j, both, have taken their tentative checkpoints
Case 4: If((m.pr_csn$_i$>csn[i]) AND (pr_csn$_j$=csn[j]));
{P_j buffers m } P_i has taken its tentative checkpoint // before sending m while P_j has not.

5.1.4 Algorithm executed at any MSS (say MSSs)

(1) Wait for Response
(2) Upon receiving message c_req (P_{in}, MSS_{in}, p_csn$_i$, m_vect) from MSS_{in}
(i)For any P_i such that mss_plist$_s$[i] =1$\wedge$ m_vect[i]=1; send c_req to P_i
(ii) ++pr_csn$_i$; mss_chk_request[i]=1, chkpt_st$_i$=1
(iii)Compute m_vect1, m_vect2, m_vect3 //Refer Section 4.1
(iv) If $\exists$ i such that m_vect1[i]=1;
send c_req to P_i. //m_vect1 contains the new processes found for the //minimum set
(3) On receiving c_req from some other MSS say MSS_p
$\forall$i such that((mss$_p$. m_vect1[i] = 1) $\wedge$ (mss_p_mss[i]= 1) $\wedge$ (mss_chk_req=1))
{ send c_req to P_i; compute m_vect1, m_vect2, m_vect3}
If $\exists$ j such that m_vect1[j]=1;
send c_req to P_j;
$\forall$i, m_vect1[i]=0;
(4) On receiving response to checkpointing from P_j
(i) If (P_j has taken the tentative checkpoint successfully the mss_chk_taken[j]=1 else mss_set fail_bit.)
(ii) If (mss_fail_bit) $\vee$ ($\forall$j mss_chk_taken[j] = mss_chk_request[j]; Send response (P_{in}, MSS_{in},mss$_s$, mss_chk_taken, mss_fail_bit, m_vect2) to MSS_{in};
(5) On receiving commit().
(i) Convert the tentative checkpoints in to permanent ones and discard old permanent checkpoints.
(ii) Process buffered messages, if any;.
(iii) $\forall$j such that m_vect[j]=1, csn[j]++;
(iv) Initialize relevant data structures.
(6) On receiving abort().
Discard the tentative checkpoints and induced checkpoints, if any.
Update relevant variables.

5.1.5 Algorithm executed at any process P_i;

On receiving tentative checkpoint request,
Take tentative checkpoint and inform local MSS.

6. Handling Node Mobility and Disconnections

An MH may be disconnected from the network for an arbitrary period of time. The Checkpointing algorithm may generate a request for such MH to take a checkpoint. Delaying a response may significantly increase the completion time of the checkpointing algorithm. We propose the following solution to deal with disconnections that may lead to infinite wait state.

When an MH, say MH_i, disconnects from an MSS, say MSS_k, MH_i takes its own checkpoint, say *disconnect_ckpt*$_i$, and transfers it to MSS_k. MSS_k stores all the relevant data structures and *disconnect_ckpt*$_i$ of MH_i on stable storage. During disconnection period, MSS_k acts on behalf of MH_i as follows. In minimum-process checkpointing, if MH_i is in the *minset[]*, *disconnect_ckpt*$_i$ is considered as MH_i's checkpoint for the current initiation. In all-process checkpointing, if MH_i's *disconnect_ckpt*$_i$ is already converted into permanent one, then the committed checkpoint is considered as the checkpoint for the current initiation; otherwise, *disconnect_ckpt*$_i$ is considered. On global checkpoint commit, MSS_k also updates MH_i's data structures, e.g., *ddv*[], *cci* etc. On the receipt of messages for MH_i, MSS_k does not update MH_i's *ddv*[] but maintains two message queues, say *old_m_q* and *new_m_q*, to store the messages as described below.

On the receipt of a message *m* for MH_i at MSS_k from any other process:
if((m.*cci*= = *cci*$_i$ $\vee$ (*m.cci*= =*nci*$_i$) $\vee$ (*matd*[j, *m.cci*]= =1))
add (*m*, *new_m_q*); // keep the message in new_m_q
else
add(*m*, old_m_q);
On all-process checkpoint commit:
Merge *new_m_q* to *old_m_q*;
Free(*new_m_q*);

When MH_i, enters in the cell of MSS_j, it is connected to the MSS_j if *g_chkpt*$_j$ is reset. Otherwise, it waits for *g_chkpt*$_j$ to be reset. Before connection, MSS_j collects MH_i's *ddv*[], *cci*, *new_m_q*, *old_m_q* from MSS_k; and MSS_k discards MH_i's support information and *disconnect_ckpt*$_i$. MSS_j sends the messages in *old_m_q* to MH_i without updating the *ddv*[], but messages in *new_m_q*, update *ddv*[] of MH_i.

6.1 Handling Failures during checkpointing

An MH may fail during checkpointing process. If an MH fails after taking its tentative checkpoint or if it is not a member of minimum set, then the checkpointing procedure can be completed uninterruptedly. If a process fails during checkpointing, then our straight forward approach is to discard the whole checkpointing operation.

The failed process will not be able to respond to the initiator's request and the initiator will detect the failure by timeout and will discard the complete checkpointing operation. If the initiator fails after sending commit, the checkpointing process can be considered complete. If the initiator fails during checkpointing, then some processes, waiting for commit will time out and will issue abort on his own.

Kim and Park [17] proposed that a process commits its tentative checkpoints if none of the processes, on which it transitively depends, fails; and the consistent recovery line is advanced for those processes that committed their checkpoints. The initiator and other processes, which transitively depend on the failed process, have to abort their tentative checkpoints. Thus, in case of a node failure during checkpointing, total abort of the checkpointing is avoided.

7. Correctness Proof

In this section, we prove that our checkpoint algorithm collects a consistent global checkpointing state. We assume that the system is in consistent state when a process initiates checkpointing.

Theorem: The global checkpointing state created by the i^{th} iteration of the checkpointing protocol is consistent.

Proof: Let global_cs$_i$ ={$C_{1,x}$, $C_{2,y}$,...........,$C_{n,z}$} be some consistent global state created by our algorithm, where $C_{i,x}$ is the x^{th} checkpoint of P_i.

The collected global checkpointing state will be inconsistent only if there is a orphan message m sent by P_i to P_j such that $C_{i,x}$ and $C_{j,y}$ are in the global state for some iteration of the checkpointing operation. We prove by contradiction that no such message exists. There are following four cases:

Case 1: $P_i \in$ m_vect[] $\wedge$ $P_j \notin$ m_vect[] (P_i belongs to the minimum set and P_j not)

As P_i has taken the permanent checkpoint in the current initiation and P_j has taken the permanent checkpoint in some previous initiation; therefore we can say that $C_{jy} \rightarrow C_{ix}$; ('$\longrightarrow$' is the Lamport's happened before relation); we have already assumed that rec(m) $\longrightarrow C_{jy}$ $\wedge$ $C_{ix} \rightarrow$ send (m)

$\Longrightarrow$ rec(m) $\longrightarrow C_{jy} \longrightarrow C_{ix} \longrightarrow$ send(m)

$\Longrightarrow$ rec (m) $\longrightarrow$ send (m)

Hence it is a contradiction.

Case 2: $P_i \in$ m_vect[] $\wedge$ $P_j \in$ m_vect[] (P_i and P_j both belong to the minimum set)

Both P_i and P_j have taken their permanent checkpoints during the current initiation; the following possibilities can take place:

P_i sends m after commit and P_j receives m before taking the tentative checkpoint. As $P_j \in$ m_vect[], the initiator MSS can issue commit only after P_j has taken its tentative checkpoint and inform the initiator. Therefore rec(m) at P_j can not take place before P_j takes its tentative checkpoint.

Suppose P_i sends m after taking the tentative checkpoint and P_j receive m before taking its tentative checkpoint. In this case, when P_j will receive m, it will check the piggybacked P_r_csn of P_i along with m and will conclude that P_i has taken tentative checkpoint for the new initiations and P_j has not taken its tentative checkpoint for this initiation. Therefore, P_j will process m only after P_j takes it tentative checkpoint. Hence the receiver of m at Pj can not occur before taking its tentative checkpointing.

Case 3: $P_i \notin$ m_vect[] $\wedge$ $P_j \in$ m_vect[] (P_j belongs to the minimum set and P_i not)

Checkpoint C_{ix} has been taken by P_i in some previous initiation and checkpoint C_{jy} has been taken by P_j in the current initiation. When P_j has taken its tentative checkpoint, it will find that P_j is dependent upon P_i and P_i is not in the minimum set computed so far. Therefore, P_j will send the c_req to P_i and P_i will be included in the minimum set. Hence it is a contradiction.

Case 4: $P_i \notin$ m_vect[] $\wedge$ $P_j \notin$ m_vect[](P_i and P_j both do not belong to the minimum set)

In this case, P_i and P_j will not take checkpoints and therefore no orphan message can exist from P_i to P_j.

Hence it is proved that no such orphan message is possible in the recorded global state collected by the proposed algorithm. Hence, the proposed algorithm leads to the consistent global state.

8. A Performance Evaluation

We compare our algorithm with the Koo and Toueg (KT) [11] algorithm, and Cao and Singhal (CS) [4] algorithm on different parameters.

(1) In CS algorithm, all processes are blocked. In the KT and the proposed algorithm only selective processes are blocked.

(2) In KT algorithm, a process is blocked, during the time, when it takes its tentative checkpoint and receives commit or abort from the initiator process.

(3) In CS algorithm, a process is blocked during the time, it sends its dependency vector to the initiator MSS and receives checkpoint request along with the minimum set.

In the proposed protocol, a process is blocked during the period, it receives m of higher CSN and it recues checkpoint request or commit message.

In CS algorithm, initiator MSS collects dependency vectors of all processes, computes minimum set and broadcasts minimum set to all MSSs. In KT algorithm and in the proposed protocol, no such step is taken.

In KT algorithm, transitive dependencies are captured by traversing direct dependencies and have a checkpoint tree is formed. It may lead to exceedingly high time for global checkpoint collection and the blocking period may also be high. In our algorithm, Transitive dependencies are captured during normal processing and hence checkpointing tree is not formed. Therefore, the time to collect the global checkpoint will be low as compared to KT algorithm. In CS algorithm, direct dependency vectors are collected in the initiation of the checkpointing algorithm. Therefore, this algorithm suffers from high synchronization message overhead.

(4) In KT algorithm and in the proposed protocol, an integer number is piggybacked onto normal messages. In CS algorithm, no such information is piggybacked onto normal messages. It can not handle the following situation. P_i receives m from P_j in the current CI such that P_j has taken some permanent checkpoint after sending m. In this case, P_i does not become causally dependent upon P_j due to receipt of m. In this case, if P_i is in the minimum set, P_j will unnecessarily be included in the minimum set.

(5) Blocking of processes takes place differently in these three protocols as follows. In KT algorithm, processes are not allowed to send any messages. In CS algorithm, processes are not allowed to send or receive any messages. In the proposed protocol, a few processes are not allowed to process the selective messages received only during the checkpointing period. A process is allowed to send messages and perform normal computations during its blocking period. It is even allowed to receive selected messages.

(6) We maintain exact dependencies among processes and a best possible knowledge of the minimum set, computed so far, at the local MSS. In this way, number of duplicate checkpoint requests is reduced as compared to the KT algorithm and no useless checkpoint requests are sent.

8.1 General Comparison with existing non-blocking minimum process algorithms:

In the algorithms [13], [19], initiator process/MSS collects dependency vectors for all the processes and computes the minimum set and sends the checkpointing request to all the processes with minimum set. These algorithms are non-blocking; the message received during checkpointing may add processes to the minimum set. It suffers from additional message overhead of sending request to all processes to send their dependency vectors and all processes send dependency vectors to the initiator process. But in our algorithm, no such overhead is imposed. The Cao-Singhal [5] suffers from the formation of checkpointing tree. In our algorithm, theoretically, we can say that the length of the checkpointing tree will be considerably low as compared to algorithm [2], as most of the transitive dependencies are captured during the normal processing. We do not compare our algorithm with Prakash-Singhal [15], as Cao-Singhal proved that there no such algorithm exists [4].

Furthermore, in algorithm [4], transitive dependencies are captured by direct dependencies. Hence the average number of useless checkpoints requests will be significantly higher than the proposed algorithm. In [5], huge data structures are piggybacked along with checkpointing request, because they are unable to maintain exact dependencies among processes. Incorrect dependencies are solved by these huge data structures. In our case, no such data structures are piggybacked on checkpointing request and no such useless checkpoint requests are sent, because we are able to maintain exact dependencies among processes and furthermore, are able to capture transitive dependencies during normal computation at the cost of piggybacking bit vector of length n for n processes onto normal computation messages.

8.2 Comparison with other Algorithms:

We use following notations to compare our algorithm with other algorithms:

N_{mss}: number of MSSs.
N_{mh}: number of MHs.
C_{pp}: cost of sending a message from one process to another
C_{st}: cost of sending a message between any two MSSs.
C_{wl}: cost of sending a message from an MH to its local MSS (or vice versa).
C_{bst}: cost of broadcasting a message over static network.
C_{search}: cost incurred to locate an MH and forward a message to its current local MSS, from a source MSS.
T_{st}: average message delay in static network.
T_{wl}: average message delay in the wireless network.
T_{ch}: average delay to save a checkpoint on the stable storage. It also includes the time to transfer the checkpoint from an MH to its local MSS.
N: total number of processes
N_{min}: number of minimum processes required to take checkpoints.
N_{mut}: number of useless mutable checkpoints [2].

T_{search}: average delay incurred to locate an MH and forward a message to its current local MSS.
N_{ucr}: average number of useless checkpoint requests in [2].
N_{dep}: average number of processes on which a process depends.
h_1: height of the checkpointing tree in Koo-Toueg algorithm [4].
h_2: height of the checkpointing tree in the proposed algorithm.:

In Koo-Toueg algorithm [4] and in the proposed one, the checkpoint initiator process, say P_{in} sends the checkpoint request to any process P_i if P_{in} is causally dependent upon P_i. Similarly, P_i sends the checkpoint request to any process P_j if P_i is causally dependent upon P_j. In this way, a checkpointing tree is formed. Theoretically, we can say that checkpointing tree will not be formed in our algorithm. But due to Z-dependencies, a low order checkpointing tree can be formed, because during normal computations all the transitive dependencies are not captured. Hence, the checkpointing tree in the proposed scheme will be negligible as compared to KT and CS algorithm in most of the practical situations.

8.3 Performance of our algorithm

8.3.1 The Synchronization message overhead:
In the first phase, a process taking a tentative checkpoint needs two system messages: request and reply. A process may receive more than one request for the same checkpoint initiation from different processes. However, we have used some techniques to reduce the duplicate checkpoint requests. Thus the system overhead is approximately $2*N_{min}*C_{pp}$ in the first phase. In the second phase, the commit requested is broadcasted on the static network; and the system overhead is C_{bst}.
***8.3.2 Number of processes taking checkpoints*:** In our algorithm, only minimum number of processes is required to take their checkpoints.

8.4 A Comparative Study

The blocking time of the Koo-Toueg [11] protocol is highest, followed by Cao-Singhal [4] algorithm. In the algorithms proposed in [5], [8], no blocking of processes takes place, but some useless checkpoints are taken, which are discarded on commit. In Elnozahy et al [8] algorithm, all processes take checkpoints. In the protocols [4], [11], and the proposed one, only minimum numbers of processes record their checkpoints. The message overhead in the proposed protocol is greater than [8], but less than [4], [5] and [11]. In algorithm [5], concurrent executions of the algorithm are allowed, but it may lead to inconsistencies in doing so [20]. We avoid concurrent executions of the proposed algorithm. In case, two processes concurrently initiate checkpointing, then the initiation of the process with lower process-ID will prevail.

	Cao-Singhal [4]	Cao-Singhal [5]	Koo-Toeg Algorithm [11]	Elnozahy et al [8]	Proposed Algorithm
Avg. blocking Time	$2T_{st}$	0	h_1*T_{ch}	0	h_2*T_{ch}
Average No. of checkpoints	N_{min}	$N_{min}+N_{mut}$	N_{min}	N	N_{min}
Average Message Overhead	$3C_{bst}+2C_{wireless}+2N_{mss}*C_{st}+3N_{mh}*C_{wl}$	$2*N_{min}*C_{pp}+C_{bst}+N_{ucr}*C_{pp}$	$3*N_{min}*C_{pp}*N_{dep}$	$2*C_{bst}+N*C_{pp}$	$2*N_{min}*C_{pp}+C_{bst}$

Table 1: A Comparison of System Performance

9. Conclusion

We have proposed a minimum process coordinated checkpointing algorithm for mobile distributed system, where no useless checkpoints are taken and an effort is made to minimize the blocking of processes. The number of processes that take checkpoints is minimized to avoid awakening of MHs in doze mode of operation and thrashing of MHs with checkpointing activity. Further, it saves limited battery life of MHs and low bandwidth of wireless channels. We have used the concept of delaying selective messages at the receiver end only during the checkpointing period. By using this technique, only selective processes are blocked for a short duration and processes are allowed to do their normal computations and send messages in the blocking period. We captured the transitive dependencies during the normal execution. The Z-dependencies are well taken care of in this protocol. We also avoided collecting dependency vectors of all processes to compute the minimum set. Thus, the proposed protocol is simultaneously able to reduce the useless checkpoints to zero and tries to optimize the blocking of processes at very less cost of maintaining exact dependencies among processes and piggybacking checkpoint sequence numbers and dependency vectors onto normal computation messages.

10. References

[1] Acharya A. and Badrinath B. R., "Checkpointing Distributed Applications on Mobile Computers," Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems, pp. 73-80, September 1994.

[2] Baldoni R., Hélary J-M., Mostefaoui A. and Raynal M., "A Communication-Induced Checkpointing Protocol that Ensures Rollback-Dependency Trackability," Proceedings of the

International Symposium on Fault-Tolerant-Computing Systems, pp. 68-77, June 1997.

[3] Cao G. and Singhal M., "On coordinated checkpointing in Distributed Systems", IEEE Transactions on Parallel and Distributed Systems, vol. 9, no.12, pp. 1213-1225, Dec 1998.

[4] Cao G. and Singhal M., "On the Impossibility of Min-process Non-blocking Checkpointing and an Efficient Checkpointing Algorithm for Mobile Computing Systems," Proceedings of International Conference on Parallel Processing, pp. 37-44, August 1998.

[5] Cao G. and Singhal M., "Mutable Checkpoints: A New Checkpointing Approach for Mobile Computing systems," IEEE Transaction On Parallel and Distributed Systems, vol. 12, no. 2, pp. 157-172, February 2001.

[6] Chandy K. M. and Lamport L., "Distributed Snapshots: Determining Global State of Distributed Systems," ACM Transaction on Computing Systems, vol. 3, No. 1, pp. 63-75, February 1985.

[7] Elnozahy E.N., Alvisi L., Wang Y.M. and Johnson D.B., "A Survey of Rollback-Recovery Protocols in Message-Passing Systems," ACM Computing Surveys, vol. 34, no. 3, pp. 375-408, 2002.

[8] Elnozahy E.N., Johnson D.B. and Zwaenepoel W., "The Performance of Consistent Checkpointing," Proceedings of the 11th Symposium on Reliable Distributed Systems, pp. 39-47, October 1992.

[9] Hélary J. M., Mostefaoui A. and Raynal M., "Communication-Induced Determination of Consistent Snapshots," Proceedings of the 28th International Symposium on Fault-Tolerant Computing, pp. 208-217, June 1998.

[10] Higaki H. and Takizawa M., "Checkpoint-recovery Protocol for Reliable Mobile Systems," Trans. of Information processing Japan, vol. 40, no.1, pp. 236-244, Jan. 1999.

[11] Koo R. and Toueg S., "Checkpointing and Roll-Back Recovery for Distributed Systems," IEEE Trans. on Software Engineering, vol. 13, no. 1, pp. 23-31, January 1987.

[12] Neves N. and Fuchs W. K., "Adaptive Recovery for Mobile Environments," Communications of the ACM, vol. 40, no. 1, pp. 68-74, January 1997.

[13] Parveen Kumar, Lalit Kumar, R K Chauhan, V K Gupta "A Non-Intrusive Minimum Process Synchronous Checkpointing Protocol for Mobile Distributed Systems" Proceedings of IEEE ICPWC-2005, pp 491-95, January 2005.

[14] Pradhan D.K., Krishana P.P. and Vaidya N.H., "Recovery in Mobile Wireless Environment: Design and Trade-off Analysis," Proceedings 26th International Symposium on Fault-Tolerant Computing, pp. 16-25, 1996.

[15] Prakash R. and Singhal M., "Low-Cost Checkpointing and Failure Recovery in Mobile Computing Systems," IEEE Transaction On Parallel and Distributed Systems, vol. 7, no. 10, pp. 1035-1048, October1996.

[16] Ssu K.F., Yao B., Fuchs W.K. and Neves N. F., "Adaptive Checkpointing with Storage Management for Mobile Environments," IEEE Transactions on Reliability, vol. 48, no. 4, pp. 315-324, December 1999.

[17] J.L. Kim, T. Park, "An efficient Protocol for checkpointing Recovery in Distributed Systems," IEEE Trans. Parallel and Distributed Systems, pp. 955-960, Aug. 1993.

[18] L. Kumar, M. Misra, R.C. Joshi, "Checkpointing in Distributed Computing Systems" Book Chapter "Concurrency in Dependable Computing", pp. 273-92, 2002.

[19] L. Kumar, M. Misra, R.C. Joshi, "Low overhead optimal checkpointing for mobile distributed systems" Proceedings. 19th IEEE International Conference on Data Engineering, pp 686 – 88, 2003.

[20] Ni, W., S. Vrbsky and S. Ray, "Pitfalls in Distributed Nonblocking Checkpointing", Journal of Interconnection Networks, Vol. 1 No. 5, pp. 47-78, March 2004.

[21] L. Lamport, "Time, clocks and ordering of events in a distributed system" Comm. ACM, vol.21, no.7, pp. 558-565, July 1978.

[22] Silva, L.M. and J.G. Silva, "Global checkpointing for distributed programs", Proc. 11th symp. Reliable Distributed Systems, pp. 155-62, Oct. 1992.

[23] Parveen Kumar, Lalit Kumar, R K Chauhan, "A Non-intrusive Hybrid Synchronous Checkpointing Protocol for Mobile Systems", IETE Journal of Research, Vol. 52 No. 2&3, 2006.

[24] Parveen Kumar, "A Low-Cost Hybrid Coordinated Checkpointing Protocol for mobile distributed systems", To appear in Mobile Information Systems.

[25] Lalit Kumar Awasthi, P.Kumar, "A Synchronous Checkpointing Protocol for Mobile Distributed Systems: Probabilistic Approach" International Journal of Information and Computer Security, Vol.1, No.3 pp 298-314.