A Low-overhead Non-Intrusive Hybrid Synchronous

Checkpointing Protocol for Mobile Systems

Parveen Kumar*
Lalit Kumar*.
R. K. chauhan**

ABSTRACT

Mobile computing raises many new issues, such as lack of stable storage, low bandwidth of wireless channels, high mobility and limited battery life. These issues make traditional checkpointing algorithms unsuitable for checkpointing mobile distributed systems. Minimum process Coordinated checkpointing is good approach to introduce fault tolerance in a distributed system transparently. This approach is domino-free and requires at most two checkpoints of each process on stable storage, and forces only interacting processes to checkpoint. Sometimes, it also requires piggybacking of information onto normal messages, blocking of the underlying computation or taking some useless checkpoints. Some of the processes that are not part of interacting set may not checkpoint for several checkpoint initiations. Thus, to take care of such processes and to get the advantage of minimum process checkpointing, we propose a non-intrusive hybrid checkpointing protocol, where an all process checkpointing is forced after the execution of minimum process checkpointing algorithm for a fixed number of times. We also optimize the information piggybacked onto each message and the number of useless checkpoints during minimum process checkpointing.

Key words **:** Fault tolerance, checkpointing, consistent global state, domino effect, coordinated checkpointing, mobile systems.

1. INTRODUCTION

A mobile distributed computing system (MDCS) is a distributed system where some processes are running on mobile hosts (MHs). An MH can change its geographical position freely from one cell to another or even to an area covered by no cell. A mobile host communicates with other nodes of the distributed system via a special node called mobile support station (MSS). A cell is a geographical area around an MSS in which it can support an MH. MSS has both wired and wireless links and acts as an interface between the static network and a part of the mobile network. Static network connects all fixed nodes in the system including the MSSs. A static node that has no support to MH can be considered as MSS with no MH.

Checkpointing can be used for fault tolerance provisioning in MDCSs. A checkpoint requires storing the state of a process on stable storage. When a fault occurs, the process rolls back and restarts from the checkpointing state. This saves all the computation done up to the last checkpointed state and only the computations done after that need to be redone. Checkpointing in distributed systems involve taking a checkpoint by all the processes or at least by a set of processes that interact with one another in performing a distributed computation. This is called consistent global state. To recover from a failure, the system restarts its execution from a previous consistent global checkpoint saved on the stable storage. A system state is said to be consistent if it contains no orphan message; i.e., a message whose receive event is recorded in the state, but its send event is lost. MDCSs raise many new issues such as mobility of nodes, 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

algorithms unsuitable for MDCSs [7].

In coordinated or synchronous checkpointing, all processes take checkpoints in such a manner that the resulting global state is consistent. Mostly the checkpointing protocol follows the two-phase commit structure [11]. In the first phase, all 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 for each process on stable storage and the recovery is very simple.

Most of the research in the area of coordinated checkpointing concentrates on minimizing synchronization messages, minimizing the number of processes that participate in recording a new global checkpoint, non-intrusiveness, minimizing the number of processes that rollback. A recent result in coordinated checkpointing states that minimizing the number of processes to checkpoint and not blocking the underlying computation during checkpointing is not possible [4]. Recently, there has been an attempt to combine minimum process checkpointing and non-blocking [15]. But this protocol may lead to inconsistencies in some situations [4], [5].

Cao and Singhal achieved both by introducing the concept of mutable checkpoints [5]. Though only minimum number of processes takes permanent checkpoints but the actual number of processes that take checkpoints is more than minimum required. The checkpoints taken by these extra processes are useless and are discarded at the end of the checkpointing algorithm. The number of useless checkpoints depends upon the duration, say uncertainty period, during which an application message can force a checkpoint on a process. In this paper, we propose a coordinated checkpointing scheme for mobile distributed systems, which is hybrid of minimum process and all process coordinated checkpointing, does not block the underlying computation during checkpointing and also optimizes the information piggybacked on each message. The number of useless induced checkpoints is also reduced by reducing the uncertainty periods of processes as in [21].

We have proposed a hybrid of minimum process and all process coordinated checkpointing protocol to checkpoint MDCSs. In minimum process checkpointing, some processes may not take checkpoints for several checkpointing intervals, and in all process-coordinated checkpointing all processes need to checkpoint for every initiation of the protocol. Where as in the former case, some of the processes may not advance their recovery line for several checkpointing intervals and in case of faults, the amount of computation lost by such processes may be too large. In the latter case the recovery line is advanced for all processes after every global checkpoint but the checkpointing overhead may be exceedingly high because all processes are forced to checkpoint. So, we have proposed a solution to these problems by forcing a all process coordinated checkpoint after fixed number of minimum process coordinated checkpoints. In the study under consideration, we have taken the all process checkpointing after every three minimum process checkpoints.

The rest of the paper is organized as follows. Section 2 presents system model. In Section 3, we describe the optimal checkpointing algorithm. The correctness proof is provided in Section 4. In section 5, we compare the proposed algorithm with existing ones. Section 6 presents conclusions.

2. System Model

Our system model is similar [5] and [II]. A mobile computing system consists of a large number of MHs and relatively fewer MSSs. An MSS has both wired and wireless links and acts as an interface between the static network and a part of the mobile network. Static network connects all MSSs. A cell is a logical or geographical area covered by an MSS. An MH can directly communicate with an MSS by a reliable FIFO wireless channel only if it is present in the cell supported by the MSS.

There are n spatially separated sequential processes denoted by P_1, P_2, ..., P_n, running on mobile hosts (MHs) and static hosts (MSSs), forming a mobile distributed computing system. The processes do no share common memory or common clock. Message passing is only way for processes to communicate with each other. Each process progresses at its own speed and messages are exchanged through reliable channels, whose transmission delays are finite but arbitrary. The messages generated by the underlying computation are referred to as computation messages or simply messages, and are denoted by mi. A process is 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 relevant information is still with this MSS. Every process is assumed to have taken a checkpoint with CI (checkpointing interval) [000] immediately before execution begins. The ith 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. The Optimal Checkpointing Algorithm

3.1 Basic Idea

Initiator process collects the direct dependencies of all processes, computes minimum set (a set of processes which need to take checkpoints along with the initiator), and sends the checkpoint request to processes in the minimum set. This will reduce the time to take a coordinated checkpoint. If new dependencies are created during checkpointing process, those are updated and a correct minimum set is formed. By doing so, number of useless checkpoints is optimized.

In order to address different checkpointing intervals (CIs), we have replaced integer csn (checkpoint sequence number) used in [4], [5] with three bits CI. We have considered only eight different checkpointing intervals and so the information piggybacked with application messages is just three bits. We have not considered CI of one bit or two bits suitable because by using one bit CI we shall be able to distinguish only two CIs, and by using two bits we shall be able to distinguish four CIs, which we assume to be insufficient. In case of three bits, we shall be able to distinguish eight different CIs and we found that it is just sufficient for all practical purposes. We assume that no message is delayed so that it reaches destination after seven CIs.

We have proposed a hybrid of minimum process and all process coordinated checkpointing protocol for checkpointing mobile distributed systems. In minimum process coordinated checkpointing, some processes may not take their checkpoints for several checkpointing intervals, and in all process coordinated checkpointing all processes need to checkpoint for every initiation of the protocol. Where as in the former case, some of the processes may not advance their recovery line for several checkpointing intervals and in case of faults, the amount of computation lost by such processes may be too large. In the latter case the recovery line is advanced for all processes after every global checkpoint but the checkpointing overhead may be exceedingly high because all processes are forced to checkpoint. So, we have proposed a solution to these problems by forcing an all process coordinated checkpoint after the execution of minimum process coordinated checkpointing algorithm for a fixed number of times. In the study under consideration, we have taken an all process global checkpoint after every three minimum process global checkpoints, which can be increased to seven without much effort.

3.2 Example

We explain our protocol with the help of an example with reference to Figure 1. Initially every process is assumed to have taken a checkpoint with CI [000]. At time t_0, P_2 initiates checkpointing process and sends request to all processes for their direct dependency vectors. At time t_1, P_2 receives the dependency vectors from all processes, computes the minimum set [which in case of Figure 1 is $\{P_2, P_3, P_4, P_6\}$]. It takes its own tentative checkpoint, updates CI and sends checkpoint request to processes in the minimum set. At time t_4, P_4 receives the checkpoint request, takes the tentative checkpoint, and updates its current CI.

During time t_3 to t_4, P_4 receives m_4 from P_5. P_4 learns that P_5 is not in the minimum set. P_4 sends checkpoint request to P_5 and also inform the initiator to include P_5 in the minimum set. However, due to very less time in the formation of minimum set, we will have very less number of such messages. On receiving the checkpoint request, P_5 takes the tentative checkpoint.

After taking the tentative checkpoint, P_4 sends the messages m10, along with its current CI [001], to P_3. P_3 takes the induced checkpoints and updates its CI before processing the messages because following conditions are met: (i) P_4 was in checkpointing state while sending the message (ii) P_3 was not in checkpointing state at the time of receiving the message (iii) P_3's next CI [001], at the time of receiving the message, is equal to P_4's CI [001] piggybacked with the message (iv) P_3 has sent at least one message i.e. m2 since last checkpoint. When P_3 gets the checkpoint request, it converts its induced checkpoint into tentative one. After taking the induced checkpoint, P_3 sends the messages m9 to P_1. P_1 takes the induced checkpoint before processing the

message for the sa above. It does not receive the checkpoint request in the current initiation and discards its induced checkpoint on receiving commit.

After taking the tentative checkpoint, P_2 sends the message m6 to P_7. P_7 updates its CI and checkpointing state before processing the message, because, above mentioned conditions, except condition (iv), are met. After receiving m6, P_7 sends the message m8 to P_6. P_6 takes the induced checkpoint before processing the message and converts it into tentative one when it gets the checkpoint request from the initiator.

At time t_2, P_2 receives responses from all relevant processes and sends the commit or abort checkpoint message to all processes. On commit, processes [which in case of figure 1 are P_1 and P_7], who have not taken any checkpoint in the current initiation or whose induced checkpoint has been discarded, also update their CIs as if they have also committed their checkpoint and their previous checkpoints are taken for the global checkpoint formation.
me reasons mentioned

checkpointing process.

(i) Each process P_i maintains the following data structures, which are preferably stored on local MSS:

cci : three bits current CI.

pci : three bits immediate last CI.

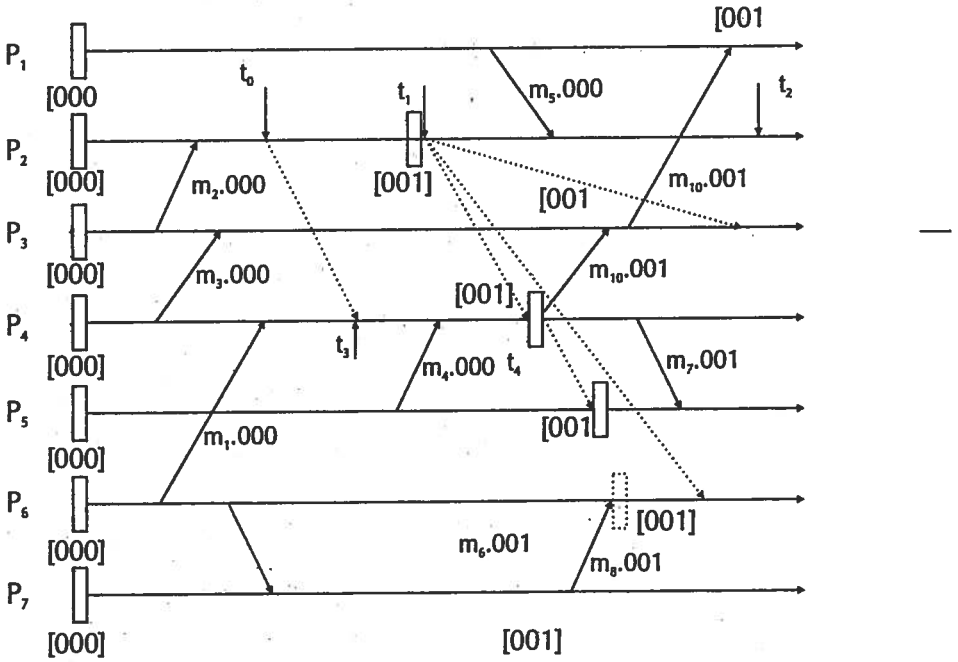
nci : three bits next CI

tentative : a flag that indicates that the process has taken the tentative checkpoint.

induced : a flag that indicates that the process has taken the induced checkpoint.

c_state : a flag, initialized to zero, set to 1 on a checkpoint.

ddv[] : a bit vector of size n. ddvi[j]=1, if P_i receives a message from Pj such that P_i becomes causally dependent upon P_j, otherwise ddvi[j]=0. Initially, the bit vector is initialized to zeroes for all processes except for itself, which is initialized to 1. For MH_i it is kept at local MSS.



Fiture 1

3.3 Data Structures

Here, we describe the data structures used in the checkpointing protocol. A process 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. Data structures are initialized on the completion of a

Send : a flag at a process; initialized to '0' on a tentative or induced checkpoint. Set to '1' when P_i sends first message after checkpoint.

(ii) Initiator MSS maintains the following Data structures:

edv[] : a bit vector of size n, computed by

taking transitive closure of ddv[] of all processes with the ddv[] of initiator process. Minimum set = {Pk such that edv[k] = 1}

R[] : is a bit vector of length n. R[i] = 1 if checkpoint response has been received from the process Pi. When the checkpoint request is sent to all MSSs, this bit vector is initialized to all zeroes.

timer; : a flag initialized to 0. Set to 1 after maximum time for collecting global checkpoint.

(iii) Each MSS (including initiator MSS) maintains the following data structures:

D[] : bit vector of length n. D[i] = 1 implies process Pi is running in the cell of MSS.

E[] : is a bit vector of length n. E[i] = 1 implies checkpoint request has been sent to process Pi in its cell. Initialized to all zeroes when MSS switches to checkpointing state.

EE[] : is a bit vector of length n. EE[i] = 1 implies checkpoint response has been received from the process Pi. Initialized to all zeroes when MSS switches to checkpointing state.

s_bit : a flag at MSS. Initialized to '0'. Set to '1' when some relevant process in its cell fails to take the tentative checkpoint.

new_d[]: a bit vector of length n. new_d [i] = 1 implies Pi has joined minimum set. Initialized to all zeroes after completion of checkpointing process.

P_{in} : checkpoint initiator process identification.

CCI_{in} : current checkpoint interval (cci) of the initiator process.

checkpointing : a flag at MSS that indicates some of the processes in its cell are in checkpointing state. Set to '1' when checkpoint request is sent to a process running in its cell.

Initialized to '0' on the completion of the checkpointing process.

g_chkpt: a flag that indicates global checkpoint collection has been initiated and is not yet complete.

$matd_{n,4}$[] : a bit dependency matrix to determine whether a message of old CI will affect the ddv[] or not. n rows denote the n processes and 4 columns denote the 4 previous checkpointing intervals. used to determine whether a message of old CI will affect the ddv[] or not.

ci_list : a link list which can contain at most four elements. Maintains checkpointing intervals and their location in matd[].

(iv) Maintenance of Different Checkpointing Intervals

Initially for a process pci, cci and nci are [000], [000] and [001] respectively. When a process takes an induced or a tentative checkpoint or switches to checkpointing state, it sets pci = cci; cci = nci; nci = modulo 8(nci + +).

When we say that a process updates its cci, it means it updates its pci, cci and nci as described above.

On global checkpoint commit, a process, who has not switched to checkpointing state in the current initiation, also updates pci, cci and nci as mentioned above. Other processes continue with their updated CIs even if their checkpoints have been discarded on commit. On checkpoint abort, all updating on CIs are also undone. When no checkpointing process is going on, then all the processes are running in the same CI.

(v) Maintenance of matd[] and ci_list

Initially, an all process global checkpoint commit, with cci = 000, is assumed, and matd[] and ci_list are initialized by using the procedure described below.

On commit, matd[] and ci_list are updated as follows:

```
if (cci = =000  cci = =100)
 {
   for (k = 1; k < =n; k + +)
        matd[k,1] = 1;
   for (k = 1 to n)
   for (l = 2 to 4)
     matd[k,l] = 0;
   free(ci_list);
   add((cci,1), ci_list);
 }
 else if (cci = =001  cci = =101)
  {
   for (k = 1; k < =n; k + +)
     {
       matd[k,2] = 1;
       if (edv[k] = =1)  matd[k,1] = 0;
       }
     add((cci,2), ci_list);
    }
   else if (cci = =010  cci = =110)
   {
     for (k = 1; k < =n; k + +)
      {
```

```
        matd[k,3]=1;
        if(edv[k]==1) {matd[k,2]=0; matd[k,1]=0}
      }
      add((cci,3), ci_list);
    }
  else if(cci==011  cci==111)
    {
    for(k=1;k<=n;k++)
      {
        matd[k,4]=1;
        if(edv[k]==1)matd[k,3]=0;
        matd[k,2]=0; matd[k,1]=0;}
       }
        add((cci,4), ci_list);
    }
```

3.4 The Checkpointing Algorithm

Any process P_i can initiate the checkpointing process. If MH_i wants to initiate, it sends a request to its current MSS that initiates and coordinates checkpointing process on behalf of MH_i. If some global checkpoint recording is already going on, then this initiation is ignored. If the current checkpointing interval is (011) or (111) then an all process checkpointing is taken, otherwise, minimum process checkpointing is initiated.

When an MH sends an application message, it needs to first send to its local MSS over the wireless cell. The MSS can piggyback appropriate information onto the application message, and then route it to the appropriate destination. Conversely, when the MSS receives an application message to be forwarded to a local MSS, it first updates the relevant vectors that it maintains for the MH, strips all piggybacked information from the message, and then forwards it to the MH. Thus, MH sends and receives application messages that do not contain any additional information; it is only responsible for checkpointing its local state appropriately and transferring it to the MSS.

3.4.1 Minimum Process Checkpointing

The initiator MSS sends the request to all MSSs to send the ddv[] (direct dependency vectors) for their processes. On receiving the request, a MSS records the identity of the initiator process and MSS, and sends back the ddv[] of its processes. Before receiving the ddv[] of all processes, if the initiator MSS receives a request for ddv[] from some other MSS and its initiator process ID is lower than the other initiator process ID, then it discards its own initiation. Alternatively, on receiving the ddv[] vectors of all processes, the initiator MSS computes the edv[] (effective dependency vector) i.e. minimum set. Initiator process takes the tentative checkpoint and initiator MSS sends the checkpoint request to all MSSs along with edv[].

On receiving a checkpoint request, an MSS sends the checkpoint request to those processes in the minimum set which are running in its cell. When an MH receives a checkpoint request, it takes the tentative checkpoint if it has not taken the induced checkpoint to this initiation. If it has already taken the induced checkpoint to this initiation, it simply converts its induced checkpoint into tentative one.

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. After taking the checkpoint, the various data structures are updated. When a process takes any kind of checkpoint, it updates its current CI and other data structures.

On receiving the checkpoint request, a process also compares bitwise, edv[]and its own ddv[].Then it finds the processes Pk for which ddv [k]=1 and edv[k]=0. If such processes are found then, the checkpoint request is also sent to these processes and a request is also sent to initiator MSS to include these processes in the minimum set.

When a process sends a message, it appends its cci (current CI) and c_state with the it. When a process Pi receives a message from other process Pj and m.cci is not equal to ncii, then the message is processed and no checkpoint is taken. Otherwise, it means that Pj has taken a checkpoint in the current initiation before sending m. Pi checks if the following conditions are met:

1. P_j was in the checkpointing state before sending m
2. P_i has sent at least one message since last checkpoint [5]
3. P_i was not in checkpointing state while receiving m

If all of these conditions are satisfied, P_i takes an induced checkpoint before processing m.

If only conditions 1 and 3 are satisfied, Pi updates its cci, nci, pci, c_state, as if it has taken a checkpoint, before processing m. A process, who has switched Checkpointing state without taking any checkpoint, does not get the checkpoint request.

When an MSS learns that all of its processes have taken the tentative checkpoint successfully or at least one of its processes has failed to take the tentative checkpoint successfully, it sends the response message to the initiator MSS. If, after sending the response message, a MSS receives the include_relevant( ) message, and learns that there is at least one process in new_d[] which is running in its cell and it has not taken the tentative checkpoint, then the MSS requests such process to take checkpoints. It again sends the response message to initiator MSS.

Finally, initiator MSS sends commit/abort to all processes. When a process receives the commit request and it has taken the tentative checkpoint then it converts its tentative checkpoint into permanent one and discards its earlier permanent checkpoint. When a process receives the commit request, and it has not received the tentative checkpoint request, then it discards its induced checkpoint, if any.

3.4.2 All process Checkpointing

For collecting all process global checkpoint, initiator sends request to all processes to checkpoint. In case of conflict for initiation, initiation with maximum address will continue. On receiving the checkpoint request, a process takes the tentative checkpoint if it has not taken the tentative checkpoint to this initiation. A process, after taking the checkpoint or knowing its inability to take the checkpoint, informs its local MSS.

When a process sends a computation message, it appends its cci (current CI) and c_state with the message. When a process, say P_i, receives a computation message from some other process, say P_j, P_i takes the tentative checkpoint before processing the message if the condition ((m.cci = nci$_i$) && (P_i.c_state = = 0) && (m.c_state = = 1)) is true. Otherwise, it simply processes the message.

When an MSS learns that all of its processes have taken the tentative checkpoint successfully or at least one of its processes has failed to take the tentative checkpoint successfully, it sends the response message to the initiator MSS. Finally, initiator sends commit/abort to all.

3.4.3 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 the disconnected MH to take a checkpoint. Delaying a response may significantly increase the completion time of the checkpointing algorithm. On reconnection, the MH may receive buffered messages which are delayed for quite long time, violating the precondition of the protocol that no message should be delayed so that it reaches destination after seven CIs. We propose the following solution to deal with arbitrary disconnections.

When an MH, say MH_i, disconnects from an MSS, say MSS_i, It stores its own checkpoint, say disconnect_ckpti, and other support information e.g. ddv[], send, cci etc., at MSS_i. During disconnection period, MSS_i acts on behalf of MH_i as follows. If checkpointing process is initiated and MH_i is in minimum set, then MSS_i converts its disconnected checkpoint into permanent one. On global checkpoint commit, MSS_i also updates MHi's data structures e.g. send, ddv[], cci etc., as if, it is a normal process. On the receipt of messages for MH_i, MSS_i does not update MHi'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:

```
if((m.cci = = cci_i ∨ (m.cci = = nci_i) ∨ (matd[j, loc (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 is reset. Otherwise, it waits for g_chkpt to be reset. Before connection, MSS_j collects its support information from MSS_i. MH_i updates its own data structures e.g. pci, cci, nci, send, ddv[] etc. on the basis of this support information. It processes the messages in old_m_q, without updating ddv[]. It processes the messages in new_m_q, updating ddv[]. It may be desired that the messages may be delivered to the MH in the order in which they were actually received by the MSS. We can use receive sequence number and store it with the received message. We can easily process the messages from the both queues

such that they are delivered in the increasing order of receive sequence number. After this, MHi starts normal functioning. MSSi discards disconnect_ckpti, if it is committed or MHi connects to some MSS. The message for disconnected MH should not be delayed so that it reaches the MSS, which maintains its support information, after 7CIs. Thus, an MH can remain disconnected for arbitrary period of time without affecting checkpointing activity.

3.5 Formal Outline of the minimum process Algorithm:

(a) Actions Taken when P_i sends a computation message to P_j:

send(P_i, m, cci_i,c_state); $send_i = 1$;

(a) Algorithm Executed at the initiator MSS:

1. If the checkpoint initiator process runs on MH, it sends the checkpoint initiation request to its local MSS.
2. if (g_chkpt) {discard the checkpoint initiation request; inform initiator; exit;}
   // some global checkpoint recording is already going on
3. MSSin sends request to all MSSs for ddv vectors; set g_chkpt;
   //Mssin is the identity of initiator MSS
4. Wait until all ddv vectors are received;
5. On the receipt of request to send ddv vectors from some other process, say Pk:
   If(Pk.ID > P_i.ID) {discard the own checkpoint initiation; exit;}
   else {ignore the request of P_k;}
6. On the receipt of all ddv vectors:
   Compute edv[] // compute minimum set of processes
7. Send take_checkpoint(Pin, MSSin, cciin, edv[]) request to all MSSs;
   // MSSin is the local MSS of the checkpoint initiator process
8. Initiator process takes the tentative checkpoint; updates data structures e.g. c_state, cci, nci etc.;
9. Set timeri ;
10. wait for response
11. On receiving Response (Pin , MSSin, MSSs, E[], new_d, s_bit) or at timer out {timeri}:
    // MSSs is the identity of MSS sending the response
    (i) If ((timeri) (s_bit))
    {send message abort(Pin,MSSin,edv[]) to all MSSs; exit;}
    (ii) for (k = 1; k <= n; k++)
    if(E[k] = = 1) R[k] = 1;
    //R[k] = 1 implies checkpoint response from the process Pk has been received
    (iii) for (k = 1; k <= n; k++)
    if(new_d[k] = = 1) edv[k] = 1;
    // new processes are added to minimum set
12. for (k = 1; k <= n; k++)
    if(R[j]! = edv[j]) goto step 10;
    //R[j]! = edv[j] implies response from all relevant processes has not arrived yet.
13. Send message commit (Pin, MSSin, edv[]) to MSSs;

(c) Actions taken when Pi receives a message from Pj:

```
if(m.cci ! = ncii)
   receive (m);
else if ((c_state = = 0) (m.c_state = = 1) (sendi )
   {take the induced checkpoint;
    update cci,nci, c_state, send etc. ;
   }
else if ((c_state = = 0) (m.c_state = = 1) (!sendi))
    update cci, nci, c_state, etc. ;
else
    receive(m);
```

(b) Algorithm Executed at any MSS, say MSSp

1. Upon receiving a message to send ddv[] from the initiator MSS:
   send own ddv[]; set g_chkpt;
   send ddv[] for MHs and disconnected MHs in its cell;
2. Upon receiving message take_checkpoint (Pin, MSSin , cciin,edv[]) from initiator MSSi:
   (i)for (j = 1; j <= n; j++)
   if ((D[j] = = 1) (edv[j] = = 1)(E[j] = = 0))
   {send the checkpoint request to Pj;
   E[j] = 1; Set checkpointing;
   }
   (ii) if (!checkpointing) {discard the checkpoint request; go to step 4 ;}
   (iii) reset s_bit;
3. (i) for (j = 1; j <= n; j++) v
   if (E[j] = = 1)
   for (k = 1; k <= n; k++)
   if (ddvj[k] = = 1 edv[k] = = 0) new_d[k] = 1;
   //Whether any process is dependent upon relevant process and not on initiator
   (ii) for (k = 1; k <= n; k++)

if (new_d[k] = = 1)
send include_relevant(MSSin, Pin, new_d[], MSSs) request to MSSs supporting the processes in new_d[];

4. Continue Computation

5 (i). Upon receiving include_relevant(MSSin, Pin, new_d[], MSSs):
for (k = 1; k < = n; k + +) n_d[k] = 0;
for (k = 1; k < = n; k + +)
{ if ((m.new_d[k] = = 1) (D[k] = = 1) (E[k] = = 0))
{E[k] = 1;
set checkpointing;
Send checkpoint request to Pk;
For (kk = 1; kk < = n; kk + +)
{if((ddvk[kk] = = 1)(edv[kk] = = 0))
n_d[kk] = 1;}
}
for (k = 1; k < = n; k + +)
if (n_d[k] = = 1)
{send include_relevant(MSSin, Pin, n_d[],MSSs) request to MSSs supporting the processes in n_d[];
for (k = 1; k < = n; k + +)
if (n_d[k] = = 1) new_d[k] = 1;

5. (ii) Upon receiving response messages from any process say Pj to which it sent checkpoint request message:
(i) set EE[j] = 1 ;
(ii) If some process in the cell has failed to take the tentative checkpoint successfully then set s_bit;

6 (i) If ((s_bit) $\vee(\forall_j$ (EE[j] = E[j])) //some relevant process has failed to checkpoint or all //relevant processes in the cell took checkpoints
{send the message Response(Pin , MSSin, MSSs, E[], s_bit, new_d[]) to initiator MSS;}
(ii) go to step 4;

7. Upon receiving abort(Pin, MSSin, edv[]) or commit (Pin,MSSin) request from initiator MSS:
send the request to all processes in its cell;

(c) 4. Algorithm Executed at any process P_i:

(a) Upon receiving a checkpoint request:
(i) if (induced) convert induced checkpoint into tentative one;
else Take a tentative checkpoint ;
(ii) Send the response to local MSS;

(a) On receiving commit:
If (tentative) {
Make_permanent (tentative_checkpoint);
Discard (old permanent checkpoint);}
else If (induced) {
Discard induced checkpoint;}
else if (c_state = = 0)
{update cci, nci, pci;)

(b) On receiving abort:
If ((tentative) (induced))
{discard the checkpoint of current initiation;
undo the updating of cci, nci, pci; }
else if (c_state = = 1) undo the updating of cci,nci etc.;

3.7 Optimizations

Following optimizations can be applied to the proposed checkpointing protocol:

(i) 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. Only the information, which changed since last checkpoint, is transferred to MSS. The MSS can reconstruct the checkpoint of the process by updating its last checkpoint with the information sent by the MH. If, due to a cell switch, the last checkpoint is not present in the current MSS, it has to be transferred from another MSS.

(ii) If an MH, on disconnection, finds that it has not sent any message since last committed checkpoint, it may skip to take disconnect checkpoint, and its previous committed checkpoint will be consistent with the new global checkpoint collected during disconnection period. Similarly, after joining another cell, checkpointing process is initiated, and the MH finds that it has not sent any message since last disconnected checkpoint, then its disconnected checkpoint can be converted into tentative checkpoint.

5. Comparisons With Existing Schemes

Acharya and Badrinath [1] gave a first checkpointing protocol for mobile systems, which is quasi-synchronous approach. In this approach, a MH takes a local checkpoint whenever a message receipt is preceded by message sent at that MH. This algorithm has no control over checkpointing activity on MHs. The number of local checkpoints may be equal to half of the number of computation messages, which may degrade the system performance. In our scheme, there is a strict control on checkpointing activity.

Koo and Toueg [11] proposed a minimum process coordinated checkpointing scheme for distributed systems.

It requires processes to be blocked during checkpointing. Checkpointing includes the time to find the minimum relevant processes and to save the state of processes on stable storage, which may be long. Therefore, this blocking algorithm may significantly reduce the performance of the system. Our protocol is non-blocking.

In Chandy-Lamport, non-blocking coordinated checkpointing algorithm, all processes are required to take checkpoints [6] and its message complexity is O(n2). In our scheme, the message complexity is O(n).

Most of the checkpointing schemes in literature are either minimum process or non-blocking for MDCSs. Prakash et al. proposed a minimum process non-blocking checkpointing scheme for MDCSs [15]. But this protocol may lead to inconsistencies in some situations [4],[5]. Cao and Singhal [5], proposed minimum process and non-blocking checkpointing schemes for MDCSs by introducing the concept of mutable checkpoints. In their protocol, only minimum number of processes takes permanent checkpoints, but the actual number of processes that take checkpoints is more than the minimum required. In the proposed scheme we have minimized the useless induced checkpoints by crashing the height of the checkpointing tree and by reducing the uncertainty period of processes. In place of integer csn, we have appended three bits CI with every message.

6. Conclusions

In this paper, we have presented a non-intrusive checkpointing protocol, which is a hybrid of minimum process and all process schemes. An attempt has been made to reduce the period during which the processes may be forced to take additional checkpoints due to non-intrusiveness. These checkpoints may or may not be required finally. But this decision can be taken only after the global collection of the checkpoint completes. Thus, such checkpoints may be useless but are unavoidable. By reducing the time, for which induced checkpoints are taken, we have reduced the number of such useless checkpoints. We have also reduced the information, to be piggybacked with every message, from an integer to three bits.

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. Briatico D., Ciuffoletti A. and Simoncini L., "A Distributed Domino-Effect Free Recovery Algorithm," Proceedings of International Symposium on Reliable Distributed Software and database, pp. 207-215, December 1984.
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. Richard G. G. and Singhal M., "Using Logging and Asynchronous Checkpointing to Implement Recoverable Distributed Shared Memory", Proceedings of the 12th symposium on Reliable Distributed Systems, pp. 58-67, October 1993.
13. Neves N. and Fuchs W. K., "Adaptive Recovery for Mobile Environments," Communications of the ACM, vol. 40, no. 1, pp. 68-74, January 1997.
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. Tsai J., Kuo S.Y. and Wang Y.M., "Theoretical Analysis for Communication-induced Checkpointing Protocols with Rollback-dependency Trackability," IEEE Transactions on Parallel & Distributed Systems, vol. 9, no. 10, pp. 963-971, October 1998.
18. Wang Y. M., "Consistent Global Checkpoints that Contain a Given Set of Local Checkpoints," IEEE Transactions on Computers, vol. 46, no. 4, pp. 456-458, April 1997.
19. Russel D. R., "State restoration in Systems of Communicating Processes," IEEE Trans. Software Engineering, vol. 6, no. 2, pp. 183-194, March 1980.
20. Park T., Woo N., Yeom H. Y., "An Efficient Optimistic Message Logging Scheme for Recoverable Mobile Computing Systems" IEEE Trans. Mobile computing, vol. 1, no. 4, pp. 265-277, December 2002.
21. 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, January 2005.

*Dept. of CSE. National Institute of 7 echnoiogy, Hamirpur (HP), India.
**Dept of Computer Sc & Application?. K.U. Kurukshetra (HRY), India lalitdecS yahoo.com; plc223475@yahoo.com