IJCSI International Journal of Computer Science Issues, Vol. 7, Issue 3, No 3, May 2010
ISSN (Online): 1694-0784
ISSN (Print): 1694-0814

A Nonblocking Coordinated Checkpointing Algorithm for Mobile Computing Systems

Rachit Garg[1], Praveen Kumar[2]

[1]Singhania University, Department of Computer Science & Engineering, Pacheri Bari (Rajasthan), India
[2]Meerut Institute of Engineering & Technology, Department of Computer Science & Engineering, Meerut (INDIA)-125005

Abstract: A checkpoint algorithm for mobile computing systems needs to handle many new issues like: mobility, low bandwidth of wireless channels, lack of stable storage on mobile nodes, disconnections, limited battery power and high failure rate of mobile nodes. These issues make traditional checkpointing techniques unsuitable for such environments. Minimum-process coordinated checkpointing is an attractive approach to introduce fault tolerance in mobile distributed systems transparently. This approach is domino-free, requires at most two checkpoints of a process on stable storage, and forces only a minimum number of processes to checkpoint. But, it requires extra synchronization messages, blocking of the underlying computation or taking some useless checkpoints**.** In this paper, we propose a nonblocking coordinated checkpointing algorithm for mobile computing systems, which requires only a minimum number of processes to take permanent checkpoints. We reduce the message complexity as compared to the Cao-Singhal algorithm [4], while keeping the number of useless checkpoints unchanged. We also address the related issues like: failures during checkpointing, disconnections, concurrent initiations of the algorithm and maintaining exact dependencies among processes. Finally, the paper presents an optimization technique, which significantly reduces the number of useless checkpoints at the cost of minor increase in the message complexity. In coordinated checkpointing, if a single process fails to take its tentative checkpoint; all the checkpoint effort is aborted. We try to reduce this effort by taking soft checkpoints in the first phase at Mobile Hosts.

Keywords: Mobile computing, fault tolerance, distributed systems, checkpointing, and minimum-process coordinated checkpointing.

1. Introduction

Mobile Hosts (MHs) are increasingly becoming common in distributed systems due to their availability, cost, and mobile connectivity. An MH is a computer that may retain its connectivity with the rest of the distributed system through a wireless network while on move. An MH communicates with the 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. An MSS has both wired and wireless links and it acts as an interface between the static network and a part of the mobile network. Static nodes are connected by a high speed wired network [1].

A checkpoint is a local state of a process saved on the stable storage. 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. A global state is said to be "consistent" if it contains no orphan message; i.e., a message whose receive event is recorded, but its send event is lost [5]. To recover from a failure, the system restarts its execution from the previous consistent global state saved on the stable storage during fault-free execution. This saves all the computation done up to the last checkpointed state and only the computation done thereafter needs to be redone.

In coordinated or synchronous checkpointing, processes take checkpoints in such a manner that the resulting global state is consistent. Mostly it follows the two-phase commit structure [2], [5], [6], [7], [10], [15]. 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 [6]. The Chandy-Lamport [5] algorithm is the earliest non-blocking all-process coordinated checkpointing algorithm.

The existence of mobile nodes in a distributed system introduces new issues that need proper handling while designing a checkpointing algorithm for such systems [1], [4], [14], [16]. These issues are mobility, disconnections, finite power source, vulnerable to physical damage, lack of stable storage etc. Prakash and Singhal [14] proposed a nonblocking minimum-process coordinated checkpointing protocol for mobile distributed systems. They proposed that a good checkpointing protocol for mobile distributed systems should have low overheads on MHs and wireless channels; and it should avoid awakening of an MH in doze mode operation. The disconnection of an MH should not lead to infinite wait state. The algorithm should be non-intrusive and it should force minimum number of processes to take their local checkpoints. In minimum-process coordinated checkpointing algorithms, some blocking of the processes takes place [3], [10], [11],[22] or some useless checkpoints are taken [4], [15].

In minimum-process coordinated checkpointing algorithms, a process P_i takes its checkpoint only if it a member of the minimum set (a subset of interacting process). A process P_i is in the minimum set only if the checkpoint initiator process is transitively dependent upon it. 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 checkpointing interval [CI] and P_k has not taken its permanent checkpoint after sending *m*. The i^{th} CI 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.

Cao and Singhal [4] achieved non-intrusiveness in the minimum-process algorithm by introducing the concept of mutable checkpoints. Kumar and Kumar [21] proposed a minimum-process coordinated checkpointing algorithm for mobile distributed systems, where the number of useless checkpoints and the blocking of processes are reduced using a probabilistic approach. Singh and Cabillic [20] proposed a minimum-process non-intrusive coordinated checkpointing protocol for deterministic mobile systems, where anti-messages of selective messages are logged during checkpointing. Higaki and Takizawa [8], and Kumar et al [17] proposed hybrid checkpointing protocols where MHs checkpoint independently and MSSs checkpoint synchronously. Neves et al. [13] gave a time based loosely synchronized coordinated checkpointing protocol that removes the overhead of synchronization and piggybacks integer csn (checkpoint sequence number). Pradhan et al [19] had shown that asynchronous checkpointing with message logging is quite effective for checkpointing mobile systems.

In the present study, we propose a nonblocking coordinated checkpointing algorithm for mobile computing systems, which requires only a minimum number of processes to take permanent checkpoints. We reduce the message complexity as compared to [4], while keeping the number of useless checkpoints unchanged.

2. Previous Coordinated Checkpointing Algorithm

Cao and Singhal [4] achieved non-intrusiveness in minimum-process algorithm by introducing the concept of mutable checkpoints. In this algorithm, checkpoint initiator process (say P_i) sends the checkpoint request to P_j only if P_i receives m from P_j in the current CI. P_i also piggybacks $csn_i[j]$ with the checkpoint request. P_j inherits the request only if $old\_csn_j \leq csn_i[j]$. old_csn_j is the csn of the current tentative or permanent checkpoint. If P_j inherits request, it acts as follows: i) P_j takes its tentative checkpoint and propagates the request to P_k only if P_j receives m from P_k in the current CI; ii) and if P_j knows that some other process has already sent the checkpoint request to P_k and P_k is not going to inherit the current checkpoint request, then P_j does not send the checkpoint request to P_k. The decision above in point (ii) is taken on the basis of data structure, MR[], received along with the checkpoint request. If P_j does not inherit the request, it simply ignores it. This process is continued till the checkpoint request reaches all the processes on which the initiator process transitively depends. Suppose, during checkpointing process, P_1 receives m from P_2. P_1 takes its mutable checkpoint before processing m only if the following conditions are met: (i) P_2 has taken some checkpoint in the current initiation before sending m (ii) P_1 has not taken any checkpoint in the current initiation (iii) P_1 has sent at least one message since its last permanent checkpoint. If P_1 takes mutable checkpoint and is not a member of the minimum set, it discards its checkpoint on commit.

We find the following observations in [4]:

(i) In this algorithm, multiple checkpoint requests may be sent between two MSSs as follows. Let us consider mobile distributed systems with two MSSs, say MSS_1 and MSS_2; where P_1 and P_2 are in the cell of MSS_1 and P_3 and P_4 are in the cell of MSS_2. Suppose, P_1 initiates checkpointing; and P_2 and P_3 are in its dependency set; i.e., P_1 is directly dependent upon P_2 and P_3. Similarly, P_4 is in the dependency set of P_2. In the existing protocol, P_1 sends checkpoint request to P_2 and P_3. After this, P_2 sends checkpoint request to P_4. In this way two messages are sent from MSS_1 to MSS_2. Although, there should be sent only one message. There is sufficient information at MSS_1 that P_1 is transitively dependent upon P_3 and P_4.

(ii) When P_i sends the checkpoint request to P_j, following scenarios are possible: (a) P_i knows that some other process has already sent the checkpoint request to P_j (b) P_j is not in the minimum set (c) P_j discards the checkpoint request and P_j actually belongs to the minimum set.

(iii) When P_i sends a checkpoint request to P_j, it also piggybacks $csn_i[j]$ and a huge data structure MR[].

(iv) $R_i[]$ maintains direct dependencies of P_i. In this algorithm, it is possible that $R_i[j]$ equals 1 and P_i is not directly dependent upon P_j for the current CI. For exactness, it is required that $R_i[j]$=1 only if P_i is directly dependent upon P_j. Hence, exact dependencies among processes are not maintained.

The useless checkpoint requests in above point [ii] are sent, because, exact dependencies among processes are not maintained as mentioned in point [iv]. The useless checkpoint requests are taken care of by sending the sufficient information along with the checkpoint requests in point [iii].

The useless checkpoint requests and the extra piggybacked information onto checkpoint requests increase the message complexity of the algorithm [4].

3. The Proposed Checkpointing Algorithm

3.1 System Model

The system model is similar to [4], [14]. A mobile computing system consists of a large number of MHs and relatively fewer MSSs. The distributed computation we consider consists of n spatially separated sequential processes denoted by $P_0, P_1, ..., P_{n-1}$, running on fail-stop MHs or on MSSs. Each MH or MSS has one process running on it. The processes do no share common memory or common clock. Message passing is the only way for the 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 m_i or m. We assume the processes to be non-deterministic.

3.2 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 checkpointing process if not mentioned explicitly. We use the term potential checkpoint request to an MSS, if at least one process takes a checkpoint in its cell to this request. Sometimes, a process is forced to take its checkpoint before processing a particular message, called mutable checkpoint [4]; and for an MH, it is preferably stored on its local disk.

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

own_csn_i:	three bits nean integer; on switching c_state_i: $own\_csn_i=csn[i]+1$; on commit or abort : after updating csn[], $own\_csn_i=csn[i]$; csn[] and c_state are described later;
$mutable_i$:	a flag that I a a flag; set to '1' on mutable checkpoint;
$ddv_i[]$:	a bit vector of size n; $ddv_i[j]$ =1 implies P_i is directly dependent upon P_j for the current CI; $ddv_i[j]$ is set to '1' only if P_i processes m received from P_j such that $m.own\_csn \geq csn[j]$; m.own_csn is the own_csn at P_j at the time of sending m and csn[j] is P_j's recent permanent checkpoint's csn; initially for P_i, $\forall k$, $ddv_i[k]=0$ and $ddv_i[i]=1$; for MH_i it is kept at local MSS; maintenance of ddv[] is described in Section 3.4;
c_state_i	a flag; set to '1' on tentative or mutable checkpoint or on receiving m from P_j s.t. $((c\_state_i = = 0) \wedge (m.c\_state = = 1) \wedge (!send_i))$; m.c_state is the c_state of P_j at the time of sending m;

send$_i$	a flag; initialized to '0' on permanent checkpoint; set to '1' when P_i sends first message after permanent checkpoint;

ii) Initiator MSS (any MSS can be initiator MSS) maintains the following Data structures:

minset[]	a bit vector of size n; minset[k]=1 implies P_k belongs to the minimum set; initially, minset[] (subset of the minimum set) is computed by using ddv vectors maintained at the initiator MSS [Refer Section 3.3]; on receiving response() from some MSS: minset=minset$\cup$ np_minset; after receiving responses from all relevant processes, minset[] contains the exact minimum set; '$\cup$', is a operator for bitwise logical OR; np_minset is described later;
R[]:	a bit vector of length n; R[i]=1 implies P_i has taken its tentative checkpoint;
timer1:	a flag; initialized to '0' when the timer is set; set to '1' when maximum allowable time for collecting coordinated checkpoint expires;

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

D[]:	a bit vector of length n; D[i]=1 implies P_i is running in the cell of MSS; it also includes the disconnected MHs supported by this MSS;
EE[]:	a bit vector of length n; EE[i] is set to '1' if P_i is in its cell and it has taken its tentativeint. tentative checkpoint;
E[]:	a bit vector of length n; E[i] is set to '1' if checkpoint request is sent to P_i and P_i is in the cell;
s_bit:	a flag; set to '1' when some relevant process in its cell fails to take its tentative checkpoint;
P$_{in}$:	initiator process identification;
MSS$_{in}$	initiator MSS identification;
own_csn$_{in}$	own_csn of initiator process;
csn[]	an array of length n for n processes; csn[j] denotes the P_j's most recent committed checkpoint's csn; on commit, for all j, (if minset [j]==1) csn[j]++; minset[] is the exact minimum set received along with the commit request; csn[] is not updated on tentative or mutable checkpoints; we maintain one csn array for each MSS and not for each process;
tnp_minset	a bit vector of length n; it contains the new processes found for the minimum set while executing a potential checkpoint request [Refer Section 3.3];
np_minset	a bit vector of length n; it contains all new processes found for the minimum set at the MSS; on each potential checkpoint request: if (tnp_minset$\neq\phi$) np_minset= np_minset$\cup$ tnp_minset;
tminset	a bit vector of length n; tminset[k]=1 implies P_k belongs to the minimum set; it maintains the local knowledge of the minimum set; on receiving tminset, minset, tnp_minset along with c_req (checkpoint request): tminset=tminset $\cup$c_req.tminset, tminset=tminset $\cup$c_req.minset, tminset=tminset $\cup$c_req.tnp_minset; on each potential checkpoint request, tnp_minset is computed, if (tnp_minset$\neq\phi$) tminset= tminset$\cup$ tnp_minset;
chkpt	a flag; set to 1 when the MSS learns that some checkpointing process is going on;
c_req	a checkpoint request; when MSS$_{in}$ sends c_req to MSS$_p$, it piggybacks the data structures: P$_{in}$, MSS$_{in}$, own_csn$_{in}$, MSS$_p$, minset; any other MSS piggybacks tminset, tnp_minset in place of minset;

3.3 Computation of minset or tnp_minset:

Let D be the bit dependency matrix of n*n, where jth row denote the ddv[] of P_j. For making dependency matrix at an MSS, if a process, say P_k, is not in the cell of MSS, then its initial ddv[] vector is assumed. Initial ddv[] of P_k is: $\forall$i, ddv[i]=0; ddv[k]=1.

Computation of minset[]: Let P_i be the initiator process.

```
    A= ddv_i[]; minset=ddv_i[]; A=A×D;
  While (A≠minset[]) do { minset=A; A= A×D;}
```

Computation of tnp_minset:

```
A=tminset; B=tminset; B=B×D;
  While (A≠B) do { A=B; B= B×D;}
   Initialize tnp_minset;
  for(i=0;i<n;i++)
     If(A[i]==1∧tminset[i]==0) tnp_minset[i]=1;
```

MSSin initially computes the minset[] on the basis of dependencies of local processes; the minset[] thus computed is based on the direct dependencies of the local processes and it is a subset of the minimum set. Suppose, MSSin sends c_req to MSSs along with minset[] and some process (say Pk) is found at MSSs, which takes the checkpoint to this c_req. All MSSs maintains the processes of minimum set to the best of their knowledge in tminset. It is required to minimize duplicate checkpoint requests. Suppose, there exists some process (say Pl) such that Pk is directly dependent upon Pl and Pl is not in the tminset (maintained by MSSs), then MSSs sends c_req to Pl. The new processes found for the minimum set while executing a potential checkpoint request at an MSS are stored in tnp_minset. For example, in the present case: tnp_minset={P_l}. MSS$_s$ sends the c_req to P_l; P_l is stored in np_minset and it is removed from the tnp_minset. In this way, np_minset at an MSS maintains all new processes found for the minimum set while executing c_req from MSS$_{in}$ or other MSSs. 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 np_minset; 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.

3.4 Maintenance of dependencies among processes

Suppose, a process P_i receives a message *m* from P_j, where *m.own_csn* is the *own_csn* at P_j at the time of sending m. When P_i sets *c_state*, it maintains two temporary bit dependency vectors, *ddv1[]* and *ddv2[]*, of length n. These are initialized to all zeroes. The dependencies created during checkpointing process are temporarily maintained in these vectors. *ddv1[]* maintains dependencies at P_i such that P_j has taken its tentative checkpoint before sending m. These dependencies persist on completion of the checkpointing process in all cases. *ddv2[]* maintains dependencies at P_i such that P_j has not taken its tentative checkpoint before sending m. These dependencies persist on completion of the checkpointing process only if P_j is not included in the *minset[]*.

Algorithm executed at P_i on the receipt of m from P_j:

```
if (m.own_csn<=csn[j])
    receive (m);
else if  (( $c\_state_i$= =0) ∧ (m.own_csn = = csn[j]))  ddv[j]=1;
else if  ((  $c\_state_i$= =0) ∧ (m.c_state = = 1) ∧ ($send_i$ ))
     {Pi takes its mutable checkpoint before processing m;
      $own\_csn_i$++; $c\_state_i$=1; $mutable_i$=1; ddv1[j]=1;}
else if  ((  $c\_state_i$= =0) ∧ (m.c_state = = 1) ∧ (!$send_i$ ))
       {$own\_csn_i$++; $c\_state_i$=1; ddv1[j]=1;}
else if  (( $c\_state_i$= =1) ∧ (m.own_csn = = csn[j]))  ddv2[j]=1;
else if  (( $c\_state_i$= =1) ∧ (m.own_csn > csn[j]))  ddv1[j]=1;
```

On Commit or Abort, ddv vector of a process P_i is updated as follows:

Case 1. The checkpointing process is aborted.

```
          for (k= 0; k<n; k++)
{ if (ddv1[k]==1 ∨ ddv2[k]==1)   ddv[k]=1;}//all dependencies persist
```

Case 2. The checkpointing process is committed and P_i is in the minimum set.

```
          for (k=0; k<n; k++)
            { ddv[k]=0; //previos dependencies of $P_i$ are initialized
             if (ddv1[k]==1)  ddv[k]=1;
               if (ddv2[k]==1 ∧ minset[k]==0)   ddv[k]=1;}
           ddv[i]=1;
```

Case 3. The checkpointing process is committed and P_i is not in the minimum set.

```
          for (k= 0; k<n; k++)
         { if (ddv[k]==1 ∧ minset[k]==1)    ddv[k]=0;
      if (ddv1[k]==1)  ddv[k]=1;
               if (ddv2[k]==1 ∧ minset[k]==0)    ddv[k]=1;}
```

Suppose, P_i receives m_k from P_j, and becomes dependent upon it. If P_j commits its checkpoint such that send (m_k) is recorded in the checkpoint of P_j, then $ddv_i[j]$ will be set to '0'. Otherwise, $ddv_i[j]$ will remain unchanged. Hence, if all the processes take checkpoints, then all the previous dependencies will be initialized; and on the contrary, if the whole of the checkpointing procedure is aborted, then all the previous dependencies will persist.

3.5 Basic Idea

The proposed checkpointing algorithm is based on keeping track of direct dependencies of processes. The initiator MSS computes minset [subset of the minimum set] on the basis of dependencies maintained locally; and sends the checkpoint request along with the minset[] to the relevant MSSs. On receiving checkpoint request, an MSS asks concerned processes to checkpoint and computes new processes for the minimum set. By using this technique, we have tried to optimize the number of messages between MSSs. In case of example, given in Section 2, point (i), MSS_1 will send just one c_req to MSS_2 to checkpoint P_3 and P_4.

When the initiator MSS commits the checkpointing process, it sends the commit request along with the exact minimum set to all MSSs and every MSS maintains up-to-date csn[]. This enables us to maintain exact dependencies among processes. In our protocol, $ddv_i[j]$=1 only if P_i is directly dependent upon P_j in the current CI. Therefore, useless checkpoint requests, as mentioned in Section 2 point (ii), are not sent in our algorithm.

When P_i sends c_req to P_j, it also piggybacks $csn_i[j]$ [4]. When P_j receives c_req, it decides, on the basis of piggybacked $csn_i[j]$, whether c_req is useful. In our protocol, no useless c_req is sent, therefore, $csn_i[j]$ is not piggybacked onto c_req.

In algorithm [4], when a process, say P_j, takes its tentative checkpoint, it also finds the processes P_k such that P_j has received m from P_k in the current CI. On the basis of MR, received with the checkpoint request, P_j decides the following: (i) whether any process has already sent the checkpoint request to P_k (ii) whether the earlier checkpoint request to P_k is useless. In our protocol, no useless checkpoint request is sent, therefore, data structures MR[] is not piggybacked onto checkpoint requests. The decision (i) is taken on the basis of tminset, maintained at every MSS. tminset maintains the local knowledge about the minimum set. In our case, instead of MR[], tminset is piggybacked onto checkpoint requests. The size of the tminset is negligibly small as compared to MR[].

In the first phase, all the MHs take induced checkpoints. When the initiator MSS comes to know that all the processes in the minimum set have taken their mutable checkpoints successfully, it sends the request to all concerned processes to convert their mutable checkpoints into tentative ones. Finally, when initiator MSS comes to know that all concerned processes have taken their tentative checkpoints successfully, it issues commit request. In this way, if a process fails to take mutable checkpoint in the first phase, then the loss of checkpointing effort is low. If all concerned MHs take tentative checkpoints in the first phase and some process fails to take its checkpoint, then the loss of checkpointing effort will be exceedingly high.

3.6 An Example

We explain our checkpointing algorithm with the help of an example. In Figure 1, at time t_1, P_2 initiates checkpointing process. $ddv_2[1]$=1 due to m_1; and $ddv_1[4]$=1 due to m_2. On the receipt of m_0, P_2 does not set ddv_2 [3] =1, because, P_3 has taken permanent checkpoint after sending m_0. We assume that P_1 and P_2 are in the cell of the same MSS, say MSS_{in}. MSS_{in} computes minset (subset of minimum set) on the basis of ddv vectors maintained at MSS_{in}, which in case of figure 1 is {P_1, P_2, P_4}. Therefore, P_2 sends checkpoint request to P_1 and P_4. After taking its tentative checkpoint, P_1 sends m_4 to P_3. P_3 takes mutable checkpoint before processing m_4. Similarly, P_4 takes mutable checkpoint before processing m_5. When P_4 receives the checkpoint request, it finds that it has already taken the mutable checkpoint; therefore, it converts its mutable checkpoint into tentative one. P_4 also finds that it was dependent upon P_5 before taking its mutable checkpoint and P_5 is not in the minimum set. Therefore, P_4 sends checkpoint request to P_5. At time t_2, P_2 receives responses from all relevant processes and sends the commit request along with the minimum set [{P_1, P_2, P_4, P_5}] to all processes. When a process, in the minimum set, receives the commit message, converts its tentative checkpoint into permanent one. When a process, not in the minimum set, receives the commit message, it discards its mutable checkpoint, if any. For the sake of simplicity, we have explained our algorithm with two-phase scheme.

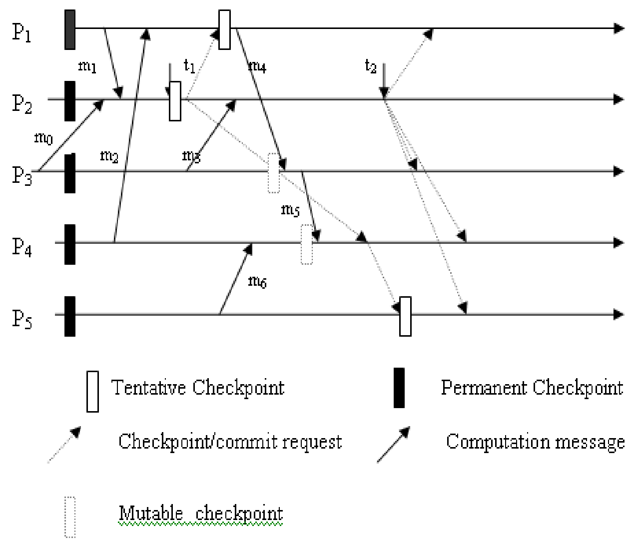

Figure 1

3.7 The Checkpointing Algorithm

Each process P_i can initiate the checkpointing process. Initiator MSS initiates and coordinates checkpointing process on behalf of MH_i. It computes minset; and sends c_req along with minset to an MSS if the later supports at least one process in the minset. It also updates its tminset on the basis of minset. We assume that concurrent invocations of the algorithm do not occur. For the sake of simplicity, we explain only two-phase protocol.

On receiving the c-req, along with the minset from the initiator MSS, an MSS, say MSS_i, takes the following actions. It updates its tminset on the basis of minset. It sends the c_req to P_i if the following conditions are met: (i) P_i is running in its cell (ii) P_i is a member of the minset and (iii) c_req has not been sent to P_i. If no such process is found, MSS_i ignores the c_req. Otherwise, on the basis of tminset, ddv vectors of processes in its cell, initial ddv vectors of other processes, it computes tnp_minset [Refer Section 3.3]. If tnp_minset is not empty, MSS_i sends c_req along with tminset, tnp_minset to an MSS, if the later supports at least one process in the tnp_minset. MSS_i updates np_minset, tminset on the basis of tnp_minset and initializes tnp_minset.

On receiving c_req along with tminset, tnp_minset from some MSS, an MSS, say MSS_j, takes the following actions. It updates its own tminset on the basis of received tminset, tnp_minset and finds any process P_k such that P_k is running in its cell, P_k has not been sent c_req and P_k is in tnp_minset. If no such process exists, it simply ignores this request. Otherwise, it sends the checkpoint request to P_k. On the basis of tminset, ddv[] of its processes and initial ddv[] of other processes, it computes tnp_minset. If tnp_minset is not empty, MSS_j sends the checkpoint request along with tminset, tnp_minset to an MSS, which supports at least one process in the tnp_minset. MSS_j updates np_minset, tminset on the basis of tnp_minset. It also initializes tnp_minset.

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. Algorithm executed at a process on the receipt of a computation message is given in Section 3.4.

When an MSS learns that all of its relevant processes have taken their tentative checkpoints successfully or at least one of its processes has failed to take its tentative checkpoint, it sends the response message along with the np_minset to the initiator MSS. If, after sending the response message, an MSS receives the checkpoint request along with the tnp_minset, and learns that there is at least one process in tnp_minset running in its cell and it has not taken its tentative checkpoint, then the MSS requests such process to take checkpoint. It again sends the response message to the initiator MSS.

When the initiator MSS receives a response from some MSS, it updates its minset on the basis of np_minset, received along with the response. Finally, initiator MSS sends commit/abort to all the processes. When a process in the minimum set receives the commit request, it converts its tentative checkpoint into permanent one and discards its earlier permanent checkpoint, if any. On receiving commit, a process discards its mutable checkpoint, if it is not a member of the minimum set.

4. Performance Evaluation

4.1 General Comparison with the Cao-Singhal Algorithm [4]:

We consider the two phases proposed algorithm for comparison with other algorithms. As mentioned in Section 2 point (ii), some useless checkpoint requests are sent in the algorithm [4]; whereas, in the proposed protocol, no such useless checkpoint requests are sent. In algorithm [4], when P_i sends checkpoint request to P_j, it also piggybacks csn_i [j] and a data structure MR. MR is an array of n pairs and each pair contains two fields: csn and r, where csn contains the csn number and r is a bit vector of length n. MR provides information to the request receivers on checkpoint request propagation decision-making. $csn_i[j]$ enables P_j to decide whether P_j inherits the request. These data structures are piggybacked onto checkpoint requests to handle useless checkpoint requests. In the proposed protocol, no useless checkpoint request is sent; therefore, there is no need to piggyback these data structures onto checkpoint requests. The $csn_i[j]$ is integer; its size is 4 bytes. In worst case the size of MR[] is (4n +n/8) bytes (n is the number of processes in the distributed system). In the proposed protocol, tminset and tnp_minset are piggybacked onto checkpoint requests. Size of each data structure is: n/8 bytes. The extra bytes piggybacked onto each checkpoint request in the algorithm [4] as compared to the proposed one are: (29n+32)/8. The number of useless checkpoint requests in [4] depends upon the number of processes, message sending rate, dependency pattern of processes etc. In some cases, the number of useless checkpoint requests in [4] may be exceedingly high. The useless checkpoint requests further increase the message complexity of the algorithm [4]. In the proposed protocol, the exact minimum set is broadcasted on the static network along with commit request, whereas in the Cao-Singhal [4] algorithm, only commit request is broadcasted. The size of the minimum set is n/8 bytes.

Concurrent executions of the algorithm are allowed in [4]. The algorithm [4] may lead to inconsistencies during its concurrent executions [15]. The proposed algorithm can be modified to allow concurrent executions on the basis of the methodology proposed in [15].

5. CONCLUSION

We have proposed a nonblocking coordinated checkpointing protocol for mobile distributed systems, where only minimum number of processes takes permanent checkpoints. We have reduced the message complexity as compared to Cao-Singhal algorithm [4], while keeping the number of useless checkpoints unchanged. The proposed algorithm is designed to impose low memory and computation overheads on MHs and low communication overheads on wireless channels. An MH can remain disconnected for an arbitrary period of time without affecting checkpointing activity. We address the issues like: failures during checkpointing, disconnections, maintaining exact dependencies among processes, and concurrent initiations. We also try to minimize the loss of checkpointing effort if some process fails to take its checkpoint in the first phase but it will increase the synchronization overhead.

REFERNCES

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) 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.
3) 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.
4) 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.
5) 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.
6) 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.
7) 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.
8) 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.
9) 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.
10) 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.
11) Parveen Kumar, R K Chauhan, "A Coordinated Checkpointing Protocol for Mobile Computing Systems", International Journal of Information and Computing Science, Vol. 9, No. 1, pp. 18-27, 2006.
12) Lalit Kumar, M. Misra, R.C. Joshi, "Low overhead optimal checkpointing for mobile distributed systems" Proceedings. 19th International Conference on IEEE Data Engineering, pp 686 – 88, 2003.
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) 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.
15) Weigang Ni, Susan V. Vrbsky and Sibabrata Ray, " Pitfalls in nonblocking checkpointing" World Science's journal of Interconnected Networks. Vol. 1 No. 5, pp. 47-78, March 2004.
16) Parveen Kumar, Lalit Kumar, R K Chauhan, "A low overhead Non-intrusive Hybrid Synchronous checkpointing protocol for mobile systems", Journal of Multidisciplinary Engineering Technologies, Vol.1, No. 1, pp 40-50, 2005.
17) Lalit Kumar, Parveen Kumar, R K chauhan "Logging based Coordinated Checkpointing in Mobile Distributed Computing Systems", IETE journal of research, vol. 51, no. 6, 2005.
18) Lamports L., "Time, clocks and ordering of events in distributed systems" Comm. ACM, 21(7), 1978, pp 558-565.
19) 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.
20) Pushpendra Singh, Gilbert Cabillic, "A Checkpointing Algorithm for Mobile Computing Environment", *LNCS, No. 2775*, pp 65-74, 2003.
21) 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, 2007.
22) Parveen Kumar, "A Low-Cost Hybrid Coordinated Checkpointing Protocol for Mobile Distributed Systems", Mobile Information Systems pp 13-32, Vol. 4, No. 1, 2007.