Mobile Information Systems 4 (2008) 13–32
IOS Press

A low-cost hybrid coordinated checkpointing protocol for mobile distributed systems

Parveen Kumar
Department of Computer Sc & Engineering, Asia Pacific Institute of Information Technology, Panipal (Haryana), India
Tel.: +91 0180 2620043; E-mail: pk223475@yahoo.com

Abstract. Mobile distributed systems raise new issues such as mobility, low bandwidth of wireless channels, disconnections, limited battery power and lack of reliable stable storage on mobile nodes. In minimum-process coordinated checkpointing, some processes may not checkpoint for several checkpoint initiations. In the case of a recovery after a fault, such processes may rollback to far earlier checkpointed state and thus may cause greater loss of computation. In all-process coordinated checkpointing, the recovery line is advanced for all processes but the checkpointing overhead may be exceedingly high. To optimize both matrices, the checkpointing overhead and the loss of computation on recovery, we propose a hybrid checkpointing algorithm, wherein an all-process coordinated checkpoint is taken after the execution of minimum-process coordinated checkpointing algorithm for a fixed number of times. Thus, the Mobile nodes with low activity or in doze mode operation may not be disturbed in the case of minimum-process checkpointing and the recovery line is advanced for each process after an all-process checkpoint. Additionally, we try to minimize the information piggybacked onto each computation message. For minimum-process checkpointing, we design a blocking algorithm, where no useless checkpoints are taken and an effort has been made to optimize the blocking of processes. We propose to delay selective messages at the receiver end. By doing so, processes are allowed to perform their normal computation, send messages and partially receive them during their blocking period. The proposed minimum-process blocking algorithm forces zero useless checkpoints at the cost of very small blocking.

Keywords: Fault tolerance, consistent global state, coordinated checkpointing and mobile systems

1. Introduction

In the mobile distributed system, some of the processes are running on mobile hosts (MHs). An MH communicates with other nodes of the system via a special node called mobile support station (MSS) [1]. A cell is a geographical area around an MSS in which it can support an MH. An MH can change its geographical position freely from one cell to another or even to an area covered by no cell. An MSS can have 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 static node that has no support to MH can be considered as an MSS with no MH.

Wang and Iqbal [30] describe the applications of mobile technology in healthcare. In paper [24], alternative data storage solution for mobile messaging services is provided. Location Management techniques for mobile systems are given in [11,27]. Christoph Endres [10] provides a survey of software infrastructures and frameworks for ubiquitous computing. Jayaputera and Taniar [14] propose an approach of mobile query processing when the users location moves from one Base Station to another and the queries cross multi-cells. Cooperative caching, which allows sharing and coordination of cached data among clients, is a potential technique to improve the data access performance and availability in

mobile ad hoc networks. N. Chand et al. [7] propose a utility based cache replacement policy to improve the data availability and reduce the local cache miss ratio.

A checkpoint is a local state of a process saved on 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. 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. This saves all the computation done up to the last checkpointed state and only the computation done thereafter needs to be redone. In distributed systems, checkpointing can be independent, coordinated [3,9,15,29] or quasi-synchronous [2,12]. Message Logging is also used for fault tolerance in distributed systems [25].

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 [3,9,15]. 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 last checkpointed state. 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. [9] 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 [5,9,16, 19,29]. Kumar et al. [18] proposed an all-process non-intrusive checkpointing protocol for distributed systems, where just one bit is piggybacked on normal messages. It results in extra overhead of vector transfers during checkpointing.

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 [1,5,26]. 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 [26] gave minimum-process coordinated checkpointing protocol for mobile distributed systems. In minimum-process coordinated checkpointing algorithms, only a subset of interacting process (called minimum set) is required to take checkpoints in an initiation. A process P_i is in the minimum set only if 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.

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 [26]. In minimum-process coordinated checkpointing algorithms, some blocking of the processes takes place [4,15], or some useless checkpoints are taken [5,16,19].

Acharya and Badrinath [1] gave a checkpointing protocol for mobile systems. In this approach, an MH takes a local checkpoint whenever a message receipt is preceded by the message sent at that MH. This

algorithm has no control over checkpointing activity on MHs and depends totally on communication patterns. In worst case, the number of local checkpoints taken will be equal to the number of computation messages, which may lead to high checkpointing overhead.

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. [16] 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 [15], and Cao and Singhal [4] proposed minimum-process blocking algorithms. Neves et al. [22] gave a loosely synchronized coordinated protocol that removes the overhead of synchronization. Higaki and Takizawa [13] proposed a hybrid checkpointing protocol where the mobile stations take checkpoints asynchronously and fixed ones synchronously using the algorithm [15]. Kumar and Kumar [20] 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.

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 [28]. Only the information, which changed since last checkpoint, is transferred to MSS.

In the present study, we design a hybrid coordinated checkpointing algorithm for mobile distributed systems, where an all-process checkpoint is taken after executing minimum-process algorithm for a fixed number of times. By proposing a hybrid scheme, we try to balance the checkpointing overhead and the loss of computation on recovery. We also reduce the piggybacked information onto each computation message. For minimum-process checkpointing, we propose a blocking algorithm, where processes are allowed to perform their normal computation, send messages and partially receive them during the blocking period.

The rest of the paper is organized as follows. We formulate the hybrid checkpointing algorithm in Section 2. The correctness proof is provided in Section 3. In Section 4, we evaluate the proposed scheme. Section 5 presents conclusions.

2. The proposed hybrid checkpointing algorithm

2.1. System model

Our system model is similar to [5,19]. There are n spatially separated sequential processes $P_0, P_1, \ldots, P_{n-1}$, running on MHs or MSSs, constituting a mobile distributed computing system. Each MH/MSS has one process running on it. The processes do not share memory or clock. Message passing is the 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. 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 i^{th} and $(i+1)^{\text{th}}$ checkpoint, including the i^{th} checkpoint but not the $(i+1)^{\text{th}}$ checkpoint.

When an MH sends an application message, it is first sent to its local MSS over the wireless cell. The MSS piggybacks appropriate information with the application message, and then routes it to the destination MSS or MH. When the MSS receives an application message to be forwarded to a local MH, it first updates the data structures that it maintains for the MH, strips all the piggybacked information, and then forwards the message to the MH. Thus, an 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 local MSS.

2.2. Basic idea

In minimum-process checkpointing, some processes, having low communication activity, may not be included in the minimum set for several checkpoint initiations and thus may not advance their recovery line for a long time. In the case of a recovery after a fault, this may lead to their rollback to far earlier checkpointed state and the loss of computation at such processes may be exceedingly high. Furthermore, due to scarce resources of MHs, this loss of computation may be undesirable. In all-process checkpointing, recovery line is advanced for each process after every global checkpoint but the checkpointing overhead may be exceedingly high, especially in mobile environments due to frequent checkpoints. MHs utilize the stable storage at the MSSs to store checkpoints of the MHs [1]. Thus, to balance the checkpointing overhead and the loss of computation on recovery, we design a hybrid checkpointing algorithm for mobile distributed systems, where an all-process checkpoint is taken after certain number of minimum-process checkpoints. The number of times, the minimum-process checkpointing algorithm is executed, depends on the particular application and environment and can be fine-tuned.

In coordinated checkpointing, an ever-increasing integer csn is generally piggybacked onto normal messages [9,29]. We propose a strategy to optimize the size of the csn. In order to address different checkpointing intervals, we have replaced integer csn with k-bit CI. Integer csn is monotonically increasing, each time a process takes its checkpoint, it increments its csn by 1. k-bit CI is used to serve the purpose of integer csn. The value of k can be fine-tuned. If we use p-bit CI, we will be able to distinguish only 2^p different CIs and it will be implicitly assumed that no message is delivered after $2^p - 1$ CIs. The lower limit of k is '1' which will lead to CI of '1' bit [18].

In the present study, we assume that all-process coordinated checkpoint is taken after the execution of minimum-process algorithm for seven times which requires only three-bit CI. In this case, any delay of a message that extends to more than seven CIs may cause a false checkpoint [18], i.e., it may trigger a checkpoint even if an initiator does not trigger checkpointing activity. Thus, in this algorithm, such delay needs to be avoided. The limit of maximum delay period of a message can be extended to fifteen CIs by using four-bit CI, but it will increase the information piggybacked onto each computation message by 1-bit. By using four-bit CI, we have the option of executing minimum-process algorithm for 3, 7 or 15 number of times before taking an all-process checkpoint. If we use two-bit CI, the maximum delay of a massage should not exceed three CIs, which seems to be unreasonably small in mobile systems. In this case, minimum-process algorithm needs to be executed for three times before taking an all-process checkpoint.

The minimum-process checkpointing algorithm is based on keeping track of direct dependencies of processes. Similar to [4], initiator process collects the direct dependency vectors of all processes,

computes minimum set, and sends the checkpoint request along with the minimum set to all processes. In this way, blocking time has been significantly reduced as compared to [15].

During the period, when a process sends its dependency set to the initiator and receives the minimum set, may receive some messages, which may alter its dependency set, and may add new members to the already computed minimum set. In order to keep the computed minimum set intact and to avoid useless checkpoints as in [16,19], we propose to block the processes for this period. We have classified the messages, received during the blocking period, into two types: (i) messages that alter the dependency set of the receiver process (ii) messages that do not alter the dependency set of the receiver process. The former messages need to be delayed at the receiver side. The messages of the later type can be processed normally. All processes can perform their normal computations and send messages during their blocking period. When a process buffers a message of former type, it does not process any message till it receives the minimum set so as to keep the proper sequence of messages received. When a process gets the minimum set, it takes the checkpoint, if it is in the minimum set. After this, it receives the buffered messages, if any. By doing so, blocking of processes is reduced as compared to [4].

2.3. 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.

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

- *cci*$_i$: Three-bit current checkpointing interval.
- *nci*$_i$: Three-bit next checkpointing interval. Maintenance of *cci* and *nci* is given below in point (iv). It is the next checkpointing interval, i.e., if P_i takes a new checkpoint, the new checkpointing interval will be nci$_i$.
- *tentative*$_i$: A flag that indicates that P_i has taken its tentative checkpoint for the current initiation.
- *ddv*$_i$[ ] : A bit vector of size *n*. *ddv*$_i[j]$ is set to '1' if P_i receives a message from P_j such that P_i becomes directly dependent upon P_j for the current CI. 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. On global commit, *ddv[ ]* of all processes are updated. In all-process checkpointing, each process initializes its *ddv[ ]* on tentative checkpoint. Maintenance of *ddv[ ]* is given in point (vi) below.
- *blocking*$_i$: A flag that indicates that the process is in blocking period. Set to '1' when P_i receives the *ddv[ ]* request; set to '0' on the receipt of the minimum set.
- *buffer*$_i$: A flag. Set to '1' when P_i buffers first message in its blocking period.
- *c_state*$_i$: A flag. Set to '1' on the receipt of the minimum set. Set to '0' on receiving *commit* or *abort*.

(*ii*) *Initiator MSS maintains the following Data structures:*

- *minset[ ]:* A bit vector of size n. Computed by taking transitive closure of *ddv[ ]* of all processes with the *ddv[ ]* of the initiator process [4]. Minimum set $=\{P_k$ such that *minset[k]*$=1\}$.
- $R[\,]$: A bit vector of length n. $R[I]$ is set to '1' if P_i has taken a tentative checkpoint.
- *Timer1:* A flag; set to '1' when maximum allowable time for collecting minimum-process global checkpoint expires.

Timer2: A flag; set to '1' when maximum allowable time for collecting all-process 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.

EE[ ]: A bit vector of length n. *EE[i]* is set to '1' if P_i has taken a tentative checkpoint and $D[i]$ =1.

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

P_{in} : Initiator process identification.

cci_{in} : P_{in}'s cci after it took its tentative checkpoint;

$matd_{n*8}[\,]$: A bit dependency matrix to determine whether a message of a particular CI will affect the *ddv[ ]* of receiver or not; n rows denote the n processes and eight columns denote eight CIs.

g_chkpt: A flag which is set to '1' on the receipt of (i) checkpoint request in all-process checkpointing or (ii)*ddv* [ ] request in minimum-process algorithm.

chkpt A flag which is set to 1 when the MSS receives the checkpoint request in the minimum-process algorithm.

mss_id An integer. It is unique to each MSS and cannot be null.

(*iv*) *Maintenance of Different CIs*

Initially, for a process, *cci* and *nci* are [000] and [001] respectively. When a process updates its CIs, it sets: (i) *cci*=*nci* (ii) *nci*=modulo 8(++*nci*); for simplicity, we only mention: *cci*=*nci*. When a process takes its tentative checkpoint, it updates its CIs. This updating is undone if the checkpointing process is aborted. During minimum-process checkpointing, all such processes, that are not part of the minimum-set, also update their CIs on commit. In this way, when no checkpointing process is going on, all the processes are having the same values of *cci*.

(*v*) *Maintenance of matd[ ]*

Initially, an all-process global checkpoint commit, with cci_{in}= [000], is assumed. On global checkpoint commit with cci_{in}=cci_c, *matd[ ]* is maintained as follows:

if (cci_c ==000) // all-process global checkpoint commit
{initialize *matd[ ];*
for (k= 0; k<n; k++)
matd[k,0]=1;}
else
{ **for** (k= 0; k<n; k++) // minimum-process checkpoint commit
matd[k, cci_c]=1;
if (*minset*[k]==1)
{ **for** (s=0; s<cci$_c$;s++)
matd[k, s]=0;}}

(*vi*) *Maintenance of ddv[ ]*

In this section, we describe, how the *ddv* vector of a process P_i is updated on the receipt of a message or during minimum-process checkpointing. When P_i sets its *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. On checkpoint completion, these vectors update dependencies of the process.

Suppose, P_i receives m from P_j, where *m.cci* is the *cci* at P_j at the time of sending m. *minset[ ]* is the exact minimum set received along with the checkpoint request. The dependency vectors at P_i [*ddv, ddv1* and *ddv2*] are maintained as follows:

if ($blocking_i$==0 $\wedge$ c_state_i==0) //no checkpointing going on
{ **if** (*matd*[j, *m.cci*]==1) *ddv*[j]=1; //P_i becomes dependent upon P_j after receiving m
else receive(m); // P_j has taken some permanent checkpoint after sending m; no *ddv*[ ] is
//updated
}
else if ($blocking_i$==1) receive(m); // P_i is in blocking period; no *ddv[ ]* is updated;
//selective messages are buffered during this period [Refer Section 2.4(c) and 2.5(c)]
else if (($tentative_i \wedge m.cci = cci_i$) $\vee$ ($!tentative_i \wedge m.cci = nci_i$)) *ddv1*[j]=1;
//P_j has taken its checkpoint for the current initiation before sending m
else if (*matd*[j, $m.cci$]==1) *ddv2*[j]=1; // Neither P_j has taken its checkpoint for the current //initiation nor P_j has taken any permanent checkpoint after sending m
else receive(m);
On Commit or Abort, *ddv* **vector of** P_i **is updated as follows:**
Case 1. The checkpointing process is aborted.
for (k= 0; k<n; k++)
if (*ddv1*[k]==1 $\vee$ *ddv2*[k]==1) *ddv*[k]=1;}
Case 2. The checkpointing process is committed and P$_i$is in the minimum set.
for (k=0; k<n; k++)
{*ddv*[k]=0;
if (*ddv1*[k]==1) *ddv*[k]=1;
if (*ddv2*[k]==1 $\wedge$ *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 $\wedge$ *minset*[k]==1) *ddv*[k]=0;
if (*ddv1*[k]==1) *ddv*[k]=1;
if (*ddv2*[k]==1 $\wedge$ *minset*[k]==0) *ddv*[k]=1;}

2.4. The proposed minimum-process checkpointing algorithm

(*a*) *Checkpoint initiation*

The initiator MSS sends a request to all MSSs (MSSs of the mobile system under consideration) to send the *ddv* vectors of the processes in their cells. All *ddv* vectors are at MSSs and thus no initial checkpointing messages or responses travels wireless channels. On receiving the *ddv*[ ] request, an MSS records the identity of the initiator process (say mss_id= mss_id_{in}) and initiator MSS, sends back the *ddv*[ ] of the processes in its cell, and sets *g_chkpt*. If the initiator MSS receives a request for *ddv*[ ] from some other MSS (say mss_id= mss_id_{in2}) and mss_id_{in} is lower than mss_id_{in2}, the current initiation (having mss_id= mss_id_{in}) is discarded and the new one (having mss_id= mss_id_{in2}) is continued. Similarly, if an MSS receives ddv requests from two MSSs, then it discards the request of the initiator MSS with

lower mss_id. Otherwise, on receiving *ddv* vectors of all processes, the initiator MSS computes *minset*[ ], sends checkpoint request to the initiator process and sends checkpoint request along with the *minset*[ ] to all MSSs.

(*b*) *Reception of a checkpoint request*

On receiving the checkpoint request along with the *minset*[ ], an MSS, say MSS$_j$, takes the following actions. It sends the checkpoint request to P_i only if P_i belongs to the *minset*[ ] and P_i is running in its cell. On receiving the checkpoint request, P_i takes its tentative checkpoint and informs MSS$_j$. On receiving positive response from P_i, MSS$_j$ updates cci_i, nci_i, resets $blocking_i$, and sends the buffered messages to P_i, if any. Alternatively, If P_i is not in the *minset*[ ] and P_i is in the cell of MSS$_j$, MSS$_j$ resets $blocking_i$ and sends the buffered message to P_i, if any. For a disconnected MH, that is a member of *minset*[ ], the MSS that has its disconnected checkpoint, converts its disconnected checkpoint into tentative one and updates its CIs.

(*c*) *Computation message received during checkpointing*

During blocking period, P_i processes m, received from P_j, if following conditions are met: (i) (!bufer$_i$) i.e. P_i has not buffered any message (ii) (*m.cci* != nci_i) i.e. P_j has not taken its tentative checkpoint before sending m (iii) (ddv_i[j]=1) $\vee$ (*matd*[j, *m.cci*]= 0)) i.e. P_i is already dependent upon P_j in the current CI or P_j has taken some permanent checkpoint after sending m.

Otherwise, the local MSS of P_i buffers m for the blocking period of P_i and sets $buffer_i$. On receiving messages, *ddv* vectors are updated as described in Section 2.3(vi).

(*d*) *Termination*

When an MSS learns that all of its processes in minimum set have taken their tentative checkpoints or at least one of its process has failed to checkpoint, it sends the response message to the initiator MSS.

Finally, initiator MSS sends commit or abort to all processes. On receiving abort, a process discards its tentative checkpoint, if any, and undoes the updating of data structures. On receiving commit, processes, in the *minset*[ ], convert their tentative checkpoints into permanent ones. On receiving commit or abort, all processes update their *ddv* vectors and other data structures.

2.5. Formal outline of the proposed minimum-process algorithm

(*a*) *Actions Taken when P_i sends m to P_j:*

send (P_i, m, cci_i);

(*b*) *Algorithm Executed at the initiator MSS:*

1. If the checkpoint initiator process, say P_{in}, runs on an MH, it sends the checkpoint initiation request to its local MSS, say MSS_{in}.
2. **if** (*g_chkpt*) { discard the checkpoint initiation request; inform initiator; exit}.
   // some global checkpoint recording is already going on
3. MSS_{in} sends request to all MSSs for *ddv* vectors; set g_chkpt_{in}.
4. *On the receipt of request to send ddv vectors from some other process, say P_k :*
   if (P_k.ID> P_{in}.ID) {discard P_{in}'s initiation; honor P_k's request; exit;}
   else { ignore the request of P_k ;}// it avoid concurrent initiations of the algorithm
5. *On the receipt of all ddv vectors:*

   (i) Compute *minset[ ]* // compute minimum set of processes

(ii) Send checkpoint request to P_{in}; cci_{in}=nci_{in};
(iii) Send *take_checkpoint* (P_{in}, MSS_{in}, cci_{in}, *minset*[ ]) request to all MSSs;

6. Set *timer1*;
7. Wait for response;
8. *On receiving Response* (P_{in}, MSS_{in}, MSS_s, *EE[ ], s_bit*) *or at timer out* {*timer1*}:
   // MSS_s is the identity of MSS sending the response
   // *EE[ ]* contains the processes at MSS_s which have checkpointed

   (i) **If** ((*timer1*) $\vee$ (s_bit))
   { send message *abort*(P_{in}, MSS_{in}, cci_{in}) to all MSSs; exit;}
   (ii) **for** (k= 0; k<n; k++)
   if (EE[k]= =1) R[k] =1;

9. **for** (k=0; k<n; k++)
   if (R[j] $\neq$*minset*[j]) go to step 7; //It implies response from P_j is still awaited.
10. Send message *commit* (P_{in}, MSS_{in}, cci_{in}) to all MSSs;

(c) *Actions taken when* P_i *receives* m *from* P_j*:*
if (!$blocking_i$) receive(m);
else if((!$buffer_i$)$\wedge$(*m.cci* != nci_i){
if (ddv_i[j]= =1$\vee$ (*matd*[j, *m.cci*]= 0) receive(m);}
else {buffer(m); set $buffer_i$;}
// P_i updates its *ddv[ ]* on processing m as described in Section 2.3 (vi)

(d) *Algorithm Executed at any MSS, say* MSS_s*:*
1. *Upon receiving a message to send ddv[ ] from the initiator MSS:*
   if (! *g_chkpt*)
   {Send *ddv[ ]* of all processes in its cell ;
   Set *g_chkpt*;
   for (j=0; j<n; j++)
   if (D[j]==1) set $blocking_j$;}
   else if (P_{new}.ID> P_{old}.ID)
   //P_{new} is the new initiator process; P_{old} is the earlier initiator process.
   Send *ddv[ ]* of all processes in its cell;
   else {ignore the request;}
2. *Upon receiving message take_checkpoint*(P_{in}, MSS_{in}, cci_{in}, *minset[ ]*):

   (i) **if** (*chkpt*) discard the checkpoint request; // duplicate request
   (ii) set *chkpt*; initialize *EE[ ]*; reset *s_bit*;
   (iii) **for** (j=0; j<n; j++)
   { **if** ((D[j]==1) $\wedge$ (*minset*[j]==1))
   {Send the checkpoint request to P_j; cci_j=nci_j;}
   else { reset $blocking_j$;
   send the buffered messages to P_j, if any;}
   }

3. Wait for a response to the checkpoint request;

4. *Upon receiving response to checkpoint request from* P_j :

   (i) **if** (P_j has taken the tentative checkpoint successfully)
   {*EE*[j]=1; send the buffered messages to P_j, if any;
   reset $blocking_j$;}
   else set *s_bit*;

   (ii) **if** ((*s_bit*)$\vee(\forall_j$ s.t.(*minset*[j]==1$\wedge D$[j]) *EE*[j]=1) //some relevant process has
   //failed to checkpoint or all relevant processes in the cell took checkpoints
   send the message *Response*(P_{in} , MSS_{in}, MSS_s, D[ ], *s_bit*) to MSS_{in};
   else go to step 3;

5. *On receiving Commit* ( ) *or Abort*( ):

   – send the request to all of its processes;
   – update data structures;

(*e*) *Algorithm Executed at any process* P_i*:*

1. *Upon receiving a tentative checkpoint request:*

   – Take a tentative checkpoint;
   – Send the response to local MSS;

2. *On receiving Commit*( ):
   if ($tentative_i$) {
   {discard old permanent checkpoint, if any;
   convert the tentative checkpoint into permanent one;}
3. *On receiving Abort* ( ):
   if ($tentative_i$)
   {discard the tentative checkpoint;}

2.6. All-process checkpointing

Our all process checkpointing algorithm is similar to Elnozahy et al. [8]. Initiator MSS sends request to all processes to checkpoint. On receiving the checkpoint request, a process takes the tentative checkpoint if it has not taken the checkpoint during current initiation. After taking a checkpoint, a process updates its CIs. A process, after taking its tentative checkpoint or knowing its inability to take the checkpoint, informs its local MSS.

When a process sends a computation message, it appends its *cci* with the message. When a process, say P_i, receives a computation message$m$from some other process, say P_j,P_i takes the tentative checkpoint before processing the message if *m.cci* equals nci_i; otherwise, it simply processes the message.

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

On commit, all processes convert their tentative checkpoints into permanent ones and update their data structures. For MHs, MSSs update the data structures. On abort, all processes discard their tentative checkpoints, if any, and undo the updating of data structures.

2.7. 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.

2.8. Example

We explain our minimum-process checkpointing algorithm with the help of an example. In Fig. 1, at time t_1, P_1 initiates checkpointing process and sends request to all processes for their *ddv* vectors. During the blocking time of a process, selective messages are buffered as follows. P_2 processes m_0, because, P_1 has taken permanent checkpoint after sending m_0. P_2 processes m_6, because, ddv_2 [3] is already 1 due to receive of m_3. P_2 buffers m_7, because, ddv_2 [4] is 0 due to non-receipt of any message from P_4 during current CI. P_2 buffers m_8 to keep the proper sequence of messages received. ddv_4 [5] equals 1 due to m_4, therefore, P_4 processes m_9. Similarly, P_5 processes m_{10}, because, ddv_5 [4] equals 1 due to m_5. P_5 buffers m_{13}, because, P_3 has taken a new checkpoint before sending m_{13} and P_5 has not received the checkpoint request from P_1.

At time t_2, P_1 receives the *ddv[ ]* from all processes [not shown in the figure], computes *minset[ ]* [which in case of Fig. 1 is $\{P_1, P_2, P_3\}$], sets cci_1=nci_1, sends checkpoint request along with the *minset[ ]* to all processes, and takes its own tentative checkpoint.

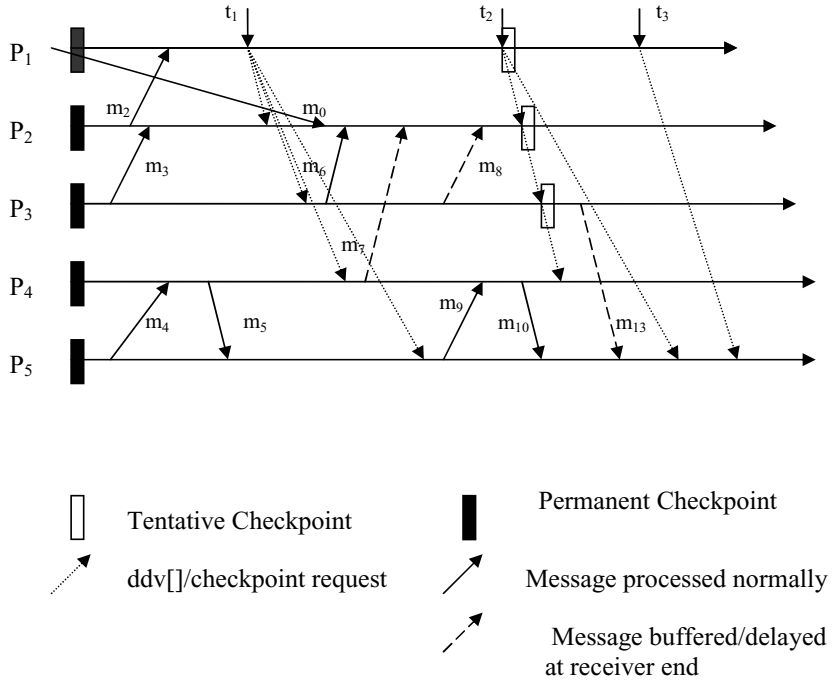

Fig. 1.

When P_2 gets the checkpoint request, it finds itself a member of the *minset [ ].* It takes the following actions: (i) take its own tentative checkpoint, (ii) set cci_2=nci_2, (iii) send the response to P_1 [not shown in the figure], (iv) process the buffered messages, i.e., m_7 and m_8. When P_5 receives the checkpoint request, it is not a member of the*minset[ ]*; therefore, it does not checkpoint but processes the buffered message, i.e., m_{13}. At time t_3, P_1 receives responses, decides to commit or abort the checkpointing activity, and sends abort or commit request to all processes.

2.9. Multiple concurrent initiations

We point out the following problems in allowing concurrent initiations in minimum-process checkpointing protocols, particularly in case of mobile distributed systems:

(i) If P_i and P_j concurrently initiate checkpointing process and P_j belongs to the minimum set of P_i, then P_j's initiation will be redundant. Some processes, in P_j's minimum set, will unnecessarily take multiple redundant checkpoints. This will waste the scarce resources of the mobile distributed system.

(ii) In case of concurrent initiations, multiple triggers need to be piggybacked on normal messages [23]. Trigger contains the initiator process identification and its csn. This leads to considerable increase in piggybacked information.

Concurrent initiations may exhaust the limited battery life and congest the wireless channels. Therefore, the concurrent executions of the proposed protocol are not considered.

2.10. Handling failures during checkpointing

Since MHs are prone to failure, an MH may fail during checkpointing process. Sudden or abrupt disconnection of an MH is also termed as a fault. Suppose, P_i is waiting for a message from P_j and P_j has failed, then P_i times out and detects the failure of P_j. If the failed process is not required to checkpoint in the current initiation or the failed process has already taken its tentative checkpoint, the checkpointing process can be completed uninterruptedly. If the failed process is not the initiator, one way to deal with the failure is to discard the whole checkpointing process similar to the approach in [15,26]. The failed process will not be able to respond to the initiator's requests and initiator will detect the failure by timeout and will abort the current checkpointing process. If the initiator fails after sending *commit* or *abort* message, it has nothing to do for the current initiation. Suppose, the initiator fails before sending *commit* or *abort* message. Some process, waiting for the checkpoint/commit request, will timeout and will detect the failure of the initiator. It will send *abort* request to all processes discarding the current checkpointing process.

The above approach seems to be inefficient, because, the whole checkpointing process is discarded even when only one participating process fails. 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.

3. Correctness proof

The correctness proof for the proposed minimum-process checkpointing algorithm is as under:

Let $\mathrm{GC}_i = \{\mathrm{C}_{1,x}, \mathrm{C}_{2,y}, \ldots, \mathrm{C}_{n,z}\}$ be some consistent global state created by our algorithm, where $C_{i,x}$ is the x^{th} checkpoint of P_i.

Theorem 1. The global state created by the i^{th} iteration of the checkpointing protocol is consistent.

Proof: Let us consider that the system is in consistent state when a process initiates checkpointing. The recorded global state will be inconsistent only if there exists a message m between two processes P_i and P_j such that P_i sends m after taking the checkpoint $C_{i,x}$, P_j receives m before taking the checkpoint $C_{j,y}$, and both $C_{i,x}$ and $C_{j,y}$ are the members of the new global state. We prove the result by contradiction that no such message exists. We consider all four possibilities as follows:

Case I: *P_i belongs to minimum set and P_j does not:*
As P_i is in minimum set, $C_{i,x}$ is the checkpoint taken by $P_i$during the current initiation and $C_{j,y}$ is the checkpoint taken by P_j during some previous initiation i.e. $C_{j,y} \rightarrow C_{i,x}$. Therefore rec(m) $\rightarrow C_{j,y}$ and $C_{i,x} \rightarrow$ send (m) implies rec(m) $\rightarrow C_{j,y} \rightarrow C_{i,x} \rightarrow$ send (m) implies rec(m) $\rightarrow$ send (m) which is not possible. '$\rightarrow$' is the Lamport's happened before relation [21].

Case II: *Both P_i and P_j are in minimum set:*
Both $C_{i,x}$ and $C_{j,y}$ are the checkpoints taken during current initiation. There are following possibilities:

(a) *P_i sends m after taking the tentative checkpoint and P_j receives m before receiving request for dependency:*
Any process can take the checkpoint only after initiator receives the dependencies from all processes. Therefore a message sent from a process after taking the checkpoint can not be received by other process before getting the dependency request.

(b) *P_i sends m after taking the tentative checkpoint and P_j receives m after getting the dependency request but before taking the checkpoint:*
In this case, following condition will be true at the time of receiving m: (*blocking*$_j$) && (*m.cci*=*nci*$_j$). Therefore, m will be buffered at P_j, and it will be processed only after P_j takes the tentative checkpoint.

(c) *P_i sends m after commit and P_j receives m before taking tentative checkpoint:*
As P_j is in the minimum set, initiator can issue a commit only after P_j takes tentative checkpoint and informs initiator. Therefore the event rec(m) at P_j cannot take place before P_j takes the checkpoint.

Case III: *P_i is not in minimum set but P_j is in minimum set:*
Checkpoint $C_{j,y}$ belongs to the current initiation and $C_{i,x}$ is from some previous initiation. The message m can be received by P_j:

(i) before receiving request for dependency
(ii) after receiving request for dependency but before taking the checkpoint $C_{j,y}$

If m is received during above (i), P_i will be included in the minimum set. If m is received during (ii) above, P_j will process m, before taking the tentative checkpoint, if any of the following conditions is true:

a. *ddv*$_j$[i]=1. In this case P_i will also be included in the minimum set.
b. (*matd*[j, *m.cci*]= 0). This is possible only if P_i has taken some permanent checkpoint after sending m. In that case, m is not an orphan message.

Case IV: *Both P_i and P_j are not in minimum set:*
Neither P_i nor P_j will take a new checkpoint, therefore, no such m is possible unless and until it already exists.

Theorem 2. Checkpointing Algorithm terminates in finite time.
Proof: When initiator initiates a new checkpoint, the initiator and other processes take the following steps:

- Initiator asks all MSSs to send the *ddv* vectors of processes in their cells. All MSSs send the same.
- Initiator computes the minimum set and sends it to all MSSs along with checkpoint request.
- All nodes that are members of minimum set take tentative checkpoints and inform the initiator. If the process is at MH, then the MH may be: disconnected, changing the cell or connected. In the first case, the disconnected checkpoint of MH is used and the last MSS converts this checkpoint to tentative on behalf of MH. In second case, the checkpoint request is delayed and MH takes the checkpoint in the new cell. In third case, MH takes the checkpoint as it is still connected. The MSS that have disconnected checkpoints or the tentative checkpoints of MHs, inform the initiator.
- After getting response from all processes/MSSs, the initiator sends commit message to all the processes.
- The processes convert their tentative checkpoints into permanent ones after receiving the commit message from the initiator.

All nodes will complete above steps in finite time unless a node is faulty. If a node in the minimum set becomes faulty during checkpointing, the whole of the checkpointing process is aborted (see Section 2.10). Hence, it can be inferred that the algorithm terminates in finite time.

Table 1
A comparison of average number of messages blocked during checkpointing

Message sending rate	0.001	0.01	0.1	1	10
Proposed algorithm	$4*10^{-7}$	$4*10^{-6}$	$4*10^{-5}$	$4*10^{-4}$	$4*10^{-3}$
Cao-Singhal algorithm [4]	$8*10^{-7}$	$8*10^{-6}$	$8*10^{-5}$	$8*10^{-4}$	$8*10^{-3}$

4. Evaluation of the protocol

Our protocol is a hybrid of all-process and minimum-process coordinated checkpointing schemes. We have also formulated a minimum-process checkpointing algorithm that can be applied independently by using integer csn in place of k-bit CI. Therefore, we evaluate our minimum-process algorithm and the hybrid algorithm separately.

4.1. Evaluation of the proposed minimum-process checkpointing algorithm

4.1.1. Computation of average blocking time and average number of messages blocked

The mobile distributed system considered has N MHs and M MSSs. Each MSS is a fixed host that has wired and wireless interface. The two MSSs are connected using a 2Mbps communication link. Each MH or MSS has one process running on it. The length of each system message is 50 bytes. The average delay on static network for sending system message is (8*50*1000)/(2*1000000) = 0.2 ms. The blocking time is 2*0.2 = 0.4 ms. In the proposed algorithm, selective incoming messages at a process are blocked during its blocking period. We consider the worst case in which all incoming messages are blocked. In Cao-Singhal [4] algorithm, a process can neither send nor receive any messages during its blocking period. The number of messages blocked at a process during its blocking period depends upon the message sending rate and blocking period and are shown in the Table 1.

The average blocking period of a message in both the algorithms is 0.4/2 = 0.2 ms. Hence, the number of messages blocked in our algorithm is less than half the number of messages blocked in the Cao-Singhal [4] algorithm, which has got the minimum blocking time of all the existing minimum-process blocking algorithms.

4.1.2. Performance of the proposed minimum-process algorithm

We use the following notations for performance analysis of the 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 [5].
N_{ind}: number of useless induced checkpoints [19].
N_{mutp}: number of useless mutable checkpoints [16].
p: number of new processes found for the minimum set after the computation of tentative minimum set at the initiator MSS [16].
h: height of the checkpointing tree in Koo-Toueg [15] algorithm.
k: number of bits used in CI in the proposed algorithm.
m: number of times the minimum-process algorithm is executed before enforcing an all-process algorithm in the proposed protocol.
T_{search}: average delay incurred to locate an MH and forward a message to its current local MSS.

The Blocking Time:

During the time, when an MSS sends the *ddv*[ ] vectors and receives the checkpoint request, all the processes in its cell remain in blocking period. During the blocking, a process can perform its normal computations, send messages and partially receive them. In worst case, blocking period of a process is $2T_{st}$.

The Synchronization message overhead:

In worst case, it includes the following:
The initiator MSS broadcasts send *ddv*[ ], take_checkpoint() and commit() messages to all MSSs: $3C_{bst}$.
The checkpoint request message from initiator process to its local MSS and its response: $2C_{\text{wireless}}$.
All MSSs send *ddv[ ]* of their processes and response to checkpoint request: $2N_{mss} * C_{st}$.
MSSs send checkpoint and commit requests to relevant processes and receive response messages: $3N_{mh} * C_{wl}$.
Total Message Overhead (say MOH_{minp}): $3C_{bst}+ 2C_{\text{wireless}}+2N_{mss}*C_{st}+ 3N_{mh}* C_{wl}$.

Number of processes taking checkpoints: In our algorithm, only minimum number of processes is required to checkpoint.

4.1.3. Comparison with other algorithms

The Koo-Toueg [15] algorithm is a minimum-process coordinated checkpointing algorithm for distributed systems. It requires processes to be blocked during checkpointing. Checkpointing includes the time to find the minimum interacting processes and to save the state of processes on stable storage, which may be too long. Therefore, this extensive blocking of processes may significantly reduce the performance of the system in mobile environments where some of the MHs may not be available due to disconnections. Each process uses monotonically increasing labels in its outgoing messages. In Koo-Toueg algorithm [15]: (i) only minimum number of processes take checkpoints (ii) message overhead is $N_{mh}*(6C_{wl} + C_{\text{search}})$ (iii) Blocking time is $N_{mh}(T_{ch} + T_{\text{search}} + 4T_{wl})$ [Refer Table 2]. Message overhead and blocking time is on significantly higher side in comparison to our minimum-process algorithm.

In Cao-Singhal algorithm [4], blocking time is reduced significantly as compared to [15]. Every process maintains direct dependencies in a bit array of length n for n processes. Initiator process collects the direct dependencies and makes a set of interacting processes (S_{forced}) which need to checkpoint along with the initiator. After sending its dependencies to the initiator and before receiving S_{forced}, a process remains in the blocking state. During blocking period, processes can do their normal computations but

Table 2
A comparison of system performance of the proposed min-process algorithm

	Cao-Singhal [4]	Cao-Singhal [5]	Koo-Toueg [15]	P. Kumar et al. [16]	Proposed min-process algorithm
Avg. blocking time	$2T_{st}$	0	$N_{mh}(T_{ch}+T_{st}+4T_{wl})$	0	$2T_{st}$
Average no. of checkpoints	N_{min}	$N_{min}+ N_{mut}$	N_{min}	$N_{min} + N_{mutp}$	N_{min}
Average message overhead	MOH_{minp}	$2*N_{min}* C_{pp} + C_{bst}$	$N_{mh}*(6C_{wl} + C_{\text{search}}$	$3C_{bst} + 2C_{wl} + (2N_{mss} + p) * C_{st} + 3N_{mh} * C_{wl}$	MOH_{minp}
Piggybacked information	Nil	Integer	Integer	Integer	Integer
Concurrent executions	No	Yes	No	No	No

cannot send any messages. The authors claim that the processes can receive messages during blocking time. The algorithm [4] is not adapted to handle the following situation. Suppose P_2 is the checkpoint initiator and it receives m from P_1 such that P_1 has taken permanent checkpoint after sending m and P_2 receives m in the current checkpointing interval before initiation. If P_1 does not send any message to any process such that P_2 becomes transitively dependent upon P_1, P_1 does not need to take its checkpoint initiated by P_2. But in the above situation P_2 will send the checkpoint request to P_1 unnecessarily. This problem arises because no information is piggybacked onto normal messages so that the receiver process can decide whether it becomes dependent upon the sender after processing the message. In our algorithm, a three-bit checkpoint sequence numbers are piggybacked onto normal messages and there is sufficient information at every MSS such that the receiver is able to maintain exact dependency information. During blocking period, processes can do their normal computations, send messages and can process selective messages. By doing so, we reduce the blocking of processes as compared to [4].

In algorithm [4]: (i) only minimum number of processes take checkpoints (ii) message overhead is $3C_{bst}+2C_{wireless}+2N_{mss}*C_{st}+3N_{mh}* C_{wl}$ (iii) Blocking time is $2T_{st}$ [Refer Table 2]. However, these parameters are similar to our algorithm. They have not piggybacked any information onto normal messages. The algorithm cannot tackle some messages as mentioned earlier in this section. In our protocol, during blocking time, processes continue their normal computation, send messages and partially receive them.

The algorithms proposed in [5,16] are non-blocking, but they suffer from useless checkpoints. The message overhead in these algorithms is also on higher side as compared to the proposed scheme.

4.2. Evaluation of the proposed hybrid algorithm

In the proposed hybrid algorithm, the all process algorithm proposed by Elnozahy et al. [9] is enforced after executing proposed min-process algorithm for m times. Therefore, the performance of the hybrid algorithm is mainly dependent upon these two algorithms and the value of m.

As shown in Table 3, the average blocking time of the Koo-Toueg [15] protocol is the highest, followed by Cao-Singhal [4] algorithm. The average blocking time of the proposed hybrid scheme is slightly less than [4]. The other schemes are non-blocking, [5,9,16,19]. In Elnozahy et al [9] algorithm, all processes take checkpoints. In the protocols [4,15], only minimum numbers of processes record their checkpoints. In non-intrusive minimum-process checkpointing scheme [5,16,19], some useless checkpoints may be taken, which are discarded on commit. The number of useless checkpoints in [19] is negligibly small as compared to [5]. In the minimum-process algorithms, some processes may starve to checkpoint and the loss of computation in the case of a recovery after a fault may be exceedingly high. In the proposed

Table 3
A comparison of system performance of the proposed hybrid protocol

	Cao-Singhal [4]	Cao- Singhal [5]	Lalit Kumar et al. [19]	Elnozahy et al. [9]	P. Kumar et al. [16]	Proposed hybrid algorithm
Avg. blocking time	$2T_{st}$	0	0	0	0	$(m*2T_{st})/(m+1)$
Average no. of checkpoints	N_{min}	$N_{min}+ N_{mut}$	$N_{min} + N_{ind}$	N	$N_{min} + N_{mutp}$	$[m * N_{min} + N]/ (m+1)$
Average message overhead	MOH_{minp}	$2*N_{min}* C_{pp}$ $+C_{bst}$	$3C_{bst} + 2C_{wl}$ $+2N_{mss} * C_{st}$ $+3N_{mh} * C_{wl}$	$2*C_{bst} + N * C_{pp}$ $[MOH_{allp}]$	$3C_{bst} + 2C_{wl}$ $+(2N_{mss} + p) * C_{st}$ $+3N_{mh} * C_{wl}$	$[m*MOH_{minp}$ $+MOH_{allp}]/(m + 1)$
Piggybacked information	Nil	Integer	Integer	Integer	Integer	k-bit [k>=1]
Concurrent executions	No	Yes	No	No	No	No

algorithm, the average number of processes that take checkpoints in an initiation is slightly greater than the minimum required; but it reduces the loss of computation on recovery.

The average message overhead in the proposed protocol is slightly less than [4,19], but greater than [9] [Refer Table 3]. In coordinated checkpointing, an integer csn is generally piggybacked on normal messages [5,9,16,19]. In the algorithm [4], no information is piggybacked on normal messages. In the proposed algorithm, k -bit CI is piggybacked on normal messages. In the present study, we have taken $k = 3$. Concurrent executions of the algorithm are allowed in [5]. W. Ni et al. [23] have shown that this algorithm [5] may lead to inconsistencies during concurrent executions.

5. Conclusion

We have designed a coordinated checkpointing algorithm which is a hybrid of minimum-process and all-process algorithms. 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. Moreover, to avoid greater loss of computation in case of a recovery after a fault, an all-process checkpoint is taken after executing minimum-process checkpointing for a fixed number of times, which, in fact, can be fine tuned. Checkpointing overhead in the proposed scheme is slightly greater than the minimum-process checkpointing but is far less than the all-process coordinated checkpointing. We have introduced the k-bit sequence numbers instead of ever increasing integer csn that is piggybacked on normal messages. This also leads to reduction in the communication overhead. We have also reduced the blocking of processes during checkpointing.

References

[1] A. Acharya and B.R. Badrinath, *Checkpointing Distributed Applications on Mobile Computers*, In Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems (PDIS 1994), 1994, 73–80.

[2] R. Baldoni, J.-M. Hélary, A. Mostefaoui and M. Raynal, *A Communication-Induced Checkpointing Protocol that Ensures Rollback-Dependency Tractability*, In Proceedings of the International Symposium on Fault-Tolerant-Computing Systems, 1997, 68–77.

[3] G. Cao and M. Singhal, On coordinated checkpointing in Distributed Systems, *IEEE Transactions on Parallel and Distributed Systems* **9**(12) (1998), 1213–1225.

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

[5] G. Cao and M. Singhal, Mutable Checkpoints: A New Checkpointing Approach for Mobile Computing systems, *IEEE Transaction On Parallel and Distributed Systems* **12**(2) (2001), 157–172.

[6] K.M. Chandy and L. Lamport, Distributed Snapshots: Determining Global State of Distributed Systems, *ACM Transaction on Computing Systems* **3**(1) (1985), 63–75.

[7] N. Chand, R.C. Joshi and M. Misra, Cooperative caching in mobile ad hoc networks based on data utility, *Mobile Information Systems* **3**(1) (2007), 19–37.

[8] E.N. Elnozahy, L. Alvisi, Y.M. Wang and D.B. Johnson, A Survey of Rollback-Recovery Protocols in Message-Passing Systems, *ACM Computing Surveys* **34**(3) (2002), 375–408.

[9] E.N. Elnozahy, D.B. Johnson and W. Zwaenepoel, *The Performance of Consistent Checkpointing*, In Proceedings of the 11th Symposium on Reliable Distributed Systems, 1992, 39–47.

[10] C. Endres, A. Butz and A. MacWilliams, A survey of software infrastructures and frameworks for ubiquitous computing, *Mobile Information Systems* **1**(1) (2005), 41–80.

[11] A. Fongen, C. Larsen, G. Ghinea, S.J.E. Taylor and T. Serif, Location based mobile computing – A tuplespace perspective, *Mobile Information Systems* **2**(2–3) (2006), 135–149.

[12] J.M. Hélary, A. Mostefaoui and M. Raynal, *Communication-Induced Determination of Consistent Snapshots*, In Proceedings of the 28$^{\text{th}}$ International Symposium on Fault-Tolerant Computing, 1998, 208–217.
[13] H. Higaki and M. Takizawa, Checkpoint-recovery Protocol for Reliable Mobile Systems, *Transactions of Information processing Japan* **40**(1) (1999), 236–244.
[14] J. Jayaputera and D. Taniar, Data retrieval for location-dependent queries in a multi-cell wireless environment, *Mobile Information Systems* **1**(2) (2005), 91–108.
[15] R. Koo and S. Toueg, Checkpointing and Roll-Back Recovery for Distributed Systems, *IEEE Transactions on Software Engineering* **13**(1) (1987), 23–31.
[16] P. Kumar, L. Kumar, R.K. Chauhan and V.K. Gupta, *A Non-Intrusive Minimum Process Synchronous Checkpointing Protocol for Mobile Distributed Systems*, In Proceedings of IEEE ICPWC-2005, 2005.
[17] J.L. Kim and T. Park, An efficient Protocol for checkpointing Recovery in Distributed Systems, *IEEE Transactions on Parallel and Distributed Systems* (1993), 955–960.
[18] L. Kumar, M. Misra and R.C. Joshi, *Checkpointing in Distributed Computing Systems,* In Concurrency in Dependable Computing, 2002, 273–292.
[19] L. Kumar, M. Misra and R.C. Joshi, *Low overhead optimal checkpointing for mobile distributed systems*, In Proceedings of 19th IEEE International Conference on Data Engineering, 2003, 686–688.
[20] L. Kumar and P.Kumar, A Synchronous Checkpointing Protocol for Mobile Distributed Systems: Probabilistic Approach, *International Journal of Information and Computer Security* **1**(3) (2007), 298–314.
[21] L. Lamport, Time, clocks and ordering of events in a distributed system, *Communications of the ACM* **21**(7) (1978), 558–565.
[22] N. Neves and W.K. Fuchs, Adaptive Recovery for Mobile Environments, *Communications of the ACM* **40**(1) (1997), 68–74.
[23] W. Ni, S. Vrbsky and S. Ray, Pitfalls in Distributed Nonblocking Checkpointing, *Journal of Interconnection Networks* **1**(5) (2004), 47–78.
[24] D.C.C. Ong, R. Sileika, S. Khaddaj and R. Oudrhiri, Alternative data storage solution for mobile messaging services, *Mobile Information Systems* **3**(1) (2007), 49–54.
[25] D.K. Pradhan, P.P. Krishana and N.H. Vaidya, *Recovery in Mobile Wireless Environment: Design and Trade-off Analysis*, In Proceedings of 26$^{\text{th}}$ International Symposium on Fault-Tolerant Computing, 1996, 16–25.
[26] R. Prakash and M. Singhal, Low-Cost Checkpointing and Failure Recovery in Mobile Computing Systems, *IEEE Transaction On Parallel and Distributed Systems* **7**(10) (1996), 1035–1048.
[27] I. Priggouris, Dimitrios Spanoudakis, Manos Spanoudakis, Stathes Hadjiefthymiades, A generic framework for Location-Based Services (LBS) provisioning, *Mobile Information Systems* **2**(2–3) (2006), 111–133.
[28] K.F. Ssu, B. Yao, W.K. Fuchs and N.F. Neves, Adaptive Checkpointing with Storage Management for Mobile Environments, *IEEE Transactions on Reliability* **48**(4) (1999), 315–324.
[29] L.M. Silva and J.G. Silva, *Global checkpointing for distributed programs*, In Proceedings of the 11$^{\text{th}}$ symposium on Reliable Distributed Systems, 1992, 155–162.
[30] X.H. Wang and M. Iqbal, Bluetooth: Opening a blue sky for healthcare, *Mobile Information Systems* **2**(2–3) (2006), 151–167.

Dr. Parveen Kumar received his Ph.D. degree in Computer Science from Kurukshetra University, Kurukshetra, India, in 2006. He has contributed over 20 technical papers in areas including checkpointing algorithms, mobile computing and distributed systems. He is serving in APIIT Panipal (India) as Professor-cum-Director (Research).