A Nonblocking Coordinated Checkpointing Algorithm for Mobile Computing Systems

Rachit Garg¹, Praveen Kumar²

¹Singhania University, Department of Computer Science & Engineering, Pacheri Bari (Rajasthan), India
²Meerut 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 Pᵢ takes its checkpoint only if it a member of the minimum set (a subset of interacting process). A process Pᵢ is in the minimum set only if the checkpoint initiator process is transitively dependent upon it. Pᵢ is directly dependent upon Pⱼ only if there exists m such that Pᵢ receives m from Pⱼ in the current checkpointing interval [CI] and Pⱼ has not taken its permanent checkpoint after sending m. The iᵗʰ CI of a process denotes all the computation performed between its iᵗʰ and (i+1)ᵗʰ checkpoint, including the iᵗʰ checkpoint but not the (i+1)ᵗʰ checkpoint.

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-intrusivenes in minimum-process algorithm by introducing the concept of mutable checkpoints. In this algorithm, checkpoint initiation process (say P1) sends the checkpoint request to Pj only if Pj receives m from P1 in the current CI. Pj also piggybacks csn[j] with the checkpoint request. Pj inherits the request only if old_csn < csn[j]. old_csn is the csn of the current tentative or permanent checkpoint. If Pj inherits request, it acts as follows: i) Pj takes its tentative checkpoint and propagates the request to Pk only if Pk receives m from Pj in the current CI; ii) if Pj knows that some other process has already sent the checkpoint request to Pk and Pk is not going to inherit the current checkpoint request, then Pj does not send the checkpoint request to Pk. The decision above in point (ii) is taken on the basis of data structure, MR[i], received along with the checkpoint request. If Pj 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, P1 receives m from P2. P1 takes its mutable checkpoint before processing m only if the following conditions are met: i) P1 has taken some checkpoint in the current initiation before sending m (ii) P1 has not taken any checkpoint in the current initiation (iii) P1 has sent at least one message since its last permanent checkpoint. If P1 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 MSS1 and MSS2; where P1 and P2 are in the cell of MSS1 and P3 and P4 are in the cell of MSS2. Suppose, P1 initiates checkpointing; and P2 and P3 are in its dependency set; current checkpoint request, then Pj does not send the checkpoint request to Pk. After this, P2 sends checkpoint request to P4. In this way two messages are sent from MSS1 to MSS2. Although, there should be sent only one message. There is sufficient information at MSS2, that P1 is transitively dependent upon P3 and P4.

(ii) When P1 sends the checkpoint request to P3, following scenarios are possible: (a) P1 knows that some other process has already sent the checkpoint request to P3 (b) P1 is not in the minimum set (c) P1 discards the checkpoint request and Pj actually belongs to the minimum set.

(iii) When P3 sends a checkpoint request to Pj, it also piggybacks csn[j] and a huge data structure MR[j].

(iv) R[j] maintains direct dependencies of Pj. In this algorithm, it is possible that R[j] equals 1 and Pj is not directly dependent upon PJ for the current CI. For exactness, it is required that R[j]=1 only if Pj is directly dependent upon PJ. 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 P0, P1, ... Pn-1, running on fail-stop MHs or on MSSs. Each MH or MSS has one process running on it. The processes do not 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, 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 Pi maintains the following data structures, which are preferably stored on local MSS:

own_csn: three bits mean integer; on switching c_state;

own_csn=csn[i]+1; on commit or abort: after updating csn[i],

own_csn=csn[i]; csn[] and c_state are described later;

mutablei: 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 Pi is directly dependent upon Pj for the current CI; ddv[i][j] is set to ‘1’ only if Pj processes m received from Pj such that m.own_csn >= csn[j];

m.own_csn is the own_csn at Pj at the time of sending m and csn[j] is Pj’s recent permanent checkpoint’s csn; initially for Pj, ∀k, ddv[k]=0 and ddv[j][j]=1; for MH, it is kept at local MSS; maintenance of ddv[] is described in Section 3.4;

c_statei: a flag; set to ‘1’ on tentative or mutable checkpoint or on receiving m from Pj s.t.

((c_statei = 0) && (m.c_state = 1) && (!sendi))); m.c_state is the c_state of Pi at the time of sending m;

ISSN (Print): 1694-0814
ISSN (Online): 1694-0784
42
Initiator MSS (any MSS can be initiator MSS) maintains the following Data structures:

- **minset[]**: a bit vector of size \( n \); \( \text{minset}[k]=1 \) implies \( P_k \) belongs to the minimum set; initially, \( \text{minset}[] \) (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: \( \text{minset} = \text{minset} \cup \text{np_minset} \); after receiving responses from all relevant processes, \( \text{minset}[] \) contains the exact minimum set; ‘\( \cup \)’ is a operator for bitwise logical OR; \( \text{np_minset} \) is described later;

- **R[]**: a bit vector of length \( n \); \( \text{R}[i]=1 \) 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;

- **dreq.minset, tminset=tminset**: on each potential checkpoint request, \( \text{tpn_minset} \) is computed, if \( \text{tpn_minset} \neq \emptyset \);

### 3.3 Computation of minset or tnp_minset:

Let \( D \) be the bit dependency matrix of \( n \times n \), where \( j^\text{th} \) row denote the \( \text{ddv}[j] \) 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 \( \text{ddv}[k] \) vector is assumed. Initial \( \text{ddv}[j] \) of \( P_j \) is: \( \forall i, \text{ddv}[i]=0; \text{ddv}[k]=1 \).

#### Computation of minset[]:
Let \( P_i \) be the initiator process.

\[
A = \text{ddv}[]; \text{minset} = \text{ddv}[]; A = A \cup D;
\]

While \( (A \neq \text{minset}) \) do \{ \( \text{minset} = A; A = A \cup D; \} \)

#### Computation of tnp_minset:

\[
A = \text{tminset}; B = \text{tminset}; B = B \times D;
\]

While \( (A \neq B) \) do \{ \( A = B; B = B \times D; \} \)

Initialize tnp_minset; for \( i = 0; i < n; i++ \)

\[
\text{If} (A[i]=1 \land \text{minset}[i]=0) \text{ tnp_minset}[i] = 1;
\]

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 \( \text{c_req} \) to MSSs along with minset[] and some process (say \( P_k \)) is found at MSSs, which takes the checkpoint to this \( \text{c_req} \). 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 \( P_l \)) such that \( P_k \) is directly dependent upon \( P_l \) and \( P_l \) is not in the tminset (maintained by MSSs), then MSSs sends \( \text{c_req} \) to \( P_l \). The new processes found for the minimum set while executing a potential checkpoint request at an MSS are stored in tnp_minset. For example, in the present case: \( \text{np_minset} = \{P_l\}; \) MSSs sends the \( \text{c_req} \) to \( P_l; P_l \) is stored in tnp_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 \( \text{c_req} \) from MSSs 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 MSSs along with np_minset; so that MSSs may update its knowledge about minimum set and wait for the new processes before sending commit. In this way, MSSs commits 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 \( m\_\text{own_csn} \) is the \( \text{own_csn} \) at \( P_i \) at the time of sending \( m \). When \( P_i \) sets \( c\_\text{state} \), it maintains two temporary bit dependency vectors, \( \text{ddv1[]} \) and \( \text{ddv2[]} \), of length \( n \). These are initialized to all zeroes. The dependencies created during checkpointing process are temporarily maintained in these vectors. \( \text{ddv1[]} \) 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. \( \text{ddv2[]} \) maintains dependencies at \( P_i \) such that \( P_j \) has not taken its tentative checkpoint before
Case 3. The checkpointing process is committed and \( P_i \) is not in the MSSs and every MSS maintains up-to-date \( csn[] \). This enables us when the initiator MSS commits the checkpointing process, it does the following: (i) whether any process has already sent the checkpoint request to \( P_i \); (ii) whether the earlier checkpoint request to \( P_i \) 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 checkpoint, 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 \( t_0 \), \( P_1 \) initiates checkpointing process. \( d\check{d}v_1[1]=1 \) due to \( m_4 \). We receive 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.

3.5 Basic Idea
The proposed checkpointing algorithm is based on keeping track of direct dependencies of processes. The initiator MSS computes \( tminset \) [subset of the minimum set] on the basis of dependencies maintained locally; and sends the checkpoint request along with the \( tminset[] \) 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), MSS1 will send just one \( c_{\text{req}} \) to MSS2 to checkpoint \( P_1 \) and \( P_2 \).

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, \( d\check{d}v[j]=1 \) 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 \( P_j \) sends \( c_{\text{req}} \) to \( P_i \) it also piggybacks \( \text{csn}(j) \) [4]. When \( P_j \) receives \( c_{\text{req}} \), it decides, on the basis of piggybacked \( \text{csn}(j) \), whether \( c_{\text{req}} \) is useful. In our protocol, no useless \( c_{\text{req}} \) is sent, therefore, \( \text{csn}(j) \) is not piggybacked onto \( c_{\text{req}} \).

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 \( P_j \); (ii) whether the earlier checkpoint request to \( P_j \) 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 checkpoint, 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 \( MR[] \).
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 $P_i$ 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.

REFERENCES


