Anti-message Logging based Check Pointing Algorithm for Mobile Distributed Systems

Monika Nagpal, PhD.
CMJ university

Praveen Kumar, PhD.
MIET
Meerut (India)

ABSTRACT
Checkpointing is one of the commonly used techniques to provide fault tolerance in distributed systems so that the system can operate even if one or more components have failed. However, mobile computing systems are constrained by low bandwidth, mobility, lack of stable storage, frequent disconnections and limited battery life. Hence checkpointing protocols which have fewer checkpoints are preferred in mobile environment. In this paper, we propose a minimum-process coordinated Checkpointing algorithm for checkpointing deterministic distributed applications on mobile systems. We eliminate useless checkpoints as well as blocking of processes during checkpoints at the cost of logging anti-messages of very few messages during Checkpointing. We also try to minimize the loss of checkpointing effort.

1. INTRODUCTION
In deterministic systems, if two processes start in the same state, and both receive the identical sequence of inputs, they will produce the identical sequence outputs and will finish in the same state. The state of a process is thus completely determined by its starting state and by sequence of messages it has received [10, 11, 12]. Johnson and Zwaenepoel [11] proposed sender based message logging for deterministic systems, where each message is logged in volatile memory on the machine from which the message is sent. The massage log is then asynchronously written to stable storage, without delaying the computation, as part of the sender's periodic checkpoint. Johnson and Zwaenepoel [12] used optimistic message logging and checkpointing to determine the maximum recoverable state, where every received message is logged. David R. Jefferson [10] introduced the concept of anti-message. Anti-message is exactly like an original message in format and content except in one field, its sign. Two messages that are identical except for opposite signs are called anti-messages of one another. All messages sent explicitly by user programs have a positive (+) sign; and their anti-messages have a negative sign (-). Whenever a message and its anti-message occur in the same queue, they immediately annihilate one another. Thus the result of enqueueing a message may be to shorten the queue by one message rather than lengthen it by one. We depict the anti-message of m by m^{-1}.

In this paper, we propose a minimum-process coordinated Checkpointing algorithm for Checkpointing deterministic distributed applications on mobile systems. We eliminate useless checkpoints as well as blocking of processes during checkpoints at the cost of logging anti-messages of very few messages during Checkpointing. We also try to minimize the loss of checkpointing effort. Frequent aborts of checkpointing procedure may happen in mobile systems due to exhausted battery, non-voluntary disconnections of MHs, or poor wireless connectivity. Therefore, we propose that in the first phase, all concerned MHs will take ad hoc checkpoint only. In case of an MH, ad hoc checkpoint is stored on the memory of MH only. In this case, if some process fails to take checkpoint in the first phase, then MHs need to abort their ad hoc checkpoints only. In this way, we try to minimize the loss of checkpointing effort when any process fails to take its checkpoint in coordination with others.

1.1 Problems in the Existing Algorithms
Singh and Cabillic [13] proposed a checkpointing algorithm for mobile computing environments on the basis of anti-message logging. This algorithm may lead to inconsistencies as follows. In Figure 1.1, at time t_0, P_1 initiates checkpointing. Since, it has received m_1 and m_2 from P_0 and P_2, respectively, since its last permanent checkpoint C_{11}; therefore, P_1 sends checkpoint request to P_0 and P_2. When P_0 receives the checkpoint request from P_1, it finds that it has not sent any message to P_1 since its last permanent checkpoint C_{02}. Therefore, P_0 discards the checkpoint request. P_2 receives m_3 without logging its anti-message. When P_2 receives the checkpoint request from P_1, it takes its tentative checkpoint C_{23}, because, it has sent m_2 to P_1 since its last permanent checkpoint C_{22}. After taking its tentative checkpoint, P_2 finds that it has received m_3 from P_0 and P_0 has already been sent the checkpoint request; therefore, P_2 does not send the checkpoint request to P_0. In this way, {C_{02}, C_{12}, C_{23}} constitute a recovery line, where m_3 is an orphan message without its anti-message being logged at P_2. Hence, the algorithm [85] may lead to inconsistencies.

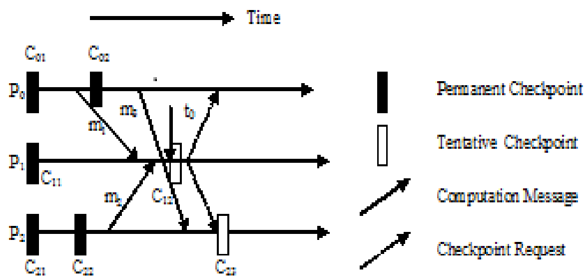

Figure 1.1. Problem in the Singh-Cabilic Alg [13]

2. THE PROPOSED CHECKPOINTING ALGORITHM
2.1 System Model
There are *n* spatially separated sequential processes denoted by P_0, P_1,.., 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. We also assume that the processes are deterministic as in [11], [12], [13].

2.2 Basic Idea

During the checkpointing procedure, a process P_i may receive m from P_j such that P_j has taken its tentative checkpoint for the current initiation whereas P_i has not taken. If P_i processes m and it receives checkpoint request later on and takes its checkpoint, then m will become orphan in the recorded global state. In order to avoid such orphan messages, Cao and Singhal [2] proposed that P_i should take a forced checkpoint before processing m. If P_i receives a checkpoint request after processing m, then the forced checkpoint already taken is converted into tentative one. In this way, m will not become orphan. P Kumar [9] proposed that such messages should be buffered at the receiver end. The receiver should process such messages only after taking its checkpoint or after getting conformed that it is not going to take its checkpoint in the current initiation. Koo-Toueg [4] proposed that P_j should not send any computation message to any process after taking its checkpoint for the current initiation. P_j starts sending messages only after getting conformed that all concerned processes have taken their checkpoint for the current initiation. We propose that the anti-messages of only those messages, which can become orphan, should be recorded at the receiver end. In deterministic systems, orphan messages are received as duplicate messages on recovery. A duplicate message is annihilated by its anti-message at the receiver end before processing. Hence, in deterministic distributed systems, an orphan message in global checkpoint does not create any inconsistency during recovery if its anti-message is logged at the receiver end. By doing so, we avoid the blocking of processes as well as the useless checkpoints in minimum-process checkpointing. It should be noted that in minimum-process coordinated checkpointing, some useless checkpoints are taken [2, 8, 14] or blocking of processes takes place [4, 6, 9]. The overheads of logging a few anti-messages may be negligible as compared to taking some useless checkpoints or blocking the processes during checkpointing.

The initiator MSS computes int_vect [subset of the minimum set] on the basis of dependencies maintained locally; and sends the checkpoint request along with the int_vect[] 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.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 comm_csn_vect[]. comm_csn_vect[] is described in Section 4.5. This enables us to maintain exact dependencies among processes. In our protocol, $cv_i[j]=1$ only if P_i is directly dependent upon P_j in the current CI. Therefore, useless checkpoint requests, as occur in [2], are not sent in our algorithm.

When P_i sends c_req to P_j, it also piggybacks $csn_i[j]$ [2]. 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 [2], 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 tint_vect, maintained at every MSS. tint_vect maintains the local knowledge about the minimum set. In our case, instead of MR[], tint_vect is piggybacked onto checkpoint requests. The size of the tint_vect is negligibly small as compared to MR[].

In coordinated checkpointing, if a single process fails to take its checkpoint; all the checkpointing effort goes waste, because, each process has to abort its tentative checkpoint. Furthermore, in order to take the tentative checkpoint, an MH needs to transfer large checkpoint data to its local MSS over wireless channels. Hence, the loss of checkpointing effort may be exceedingly high due to frequent aborts of checkpointing algorithms especially in mobile systems. In mobile distributed systems, there remain certain issues like: abrupt disconnection, exhausted battery power, or failure in wireless bandwidth. So there remains a good probability that some MH may fail to take its checkpoint in coordination with others. Therefore, we propose that in the first phase, all processes in the minimum set, take ad hoc checkpoint only. Ad hoc checkpoint is stored on the memory of MH only. If some process fails to take its checkpoint in the first phase, then other MHs need to abort their ad hoc checkpoints only. The effort of taking an ad hoc checkpoint is negligible as compared to the tentative one. In this second phase, a process converts its ad hoc checkpoint into tentative one. By using this scheme, we try to minimize the loss of checkpointing effort in case of abort of checkpointing algorithm in the first phase.

2.3 Data Structures

Here, we describe the data structures used in the checkpointing protocol. A process that initiates checkpointing, is called initiator process and its local MSS is called initiator MSS. If the initiator process is on an MSS, then the MSS is the initiator MSS. Data structures are initialized on the completion of a 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.

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

p_cn_i: an integer; it is a process csn; on tentative checkpoint: p_cn_i = comm_csn_vect[i]+1 ; on commit or abort: after updating comm_csn_vect[] , p_cn_i=comm_csn_vect[i] ; comm_csn_vect[] is described later described later ;

$cv_i[]$: $cv_i[j]=1$ implies P_i is causally dependent upon P_j. $cv_i[j]$ is set to '1' only if P_i processes m received from P_j such that m.p_cn≥ comm_csn_vect[j]; m.p_cn is the p_cn at P_j at the time of sending m and omm._csn_vect[j] is P_j's recent permanent checkpoint's omm._csn_vect; initially for P_i, $\forall$k, $cv_i[k]=0$ and $cv_i[i]=1$; for MH_i it is kept at local MSS; maintenance of cv[] is described in Section

4.5.2;

$tentative_i$ a flag; set to '1' on tentative checkpoint;

$adhoc_i$ a flag; set to '1' on ad hoc checkpoint;

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

int_vect[] a bit vector of size n; int_vect[k]=1 implies P_k belongs to the minimum set; initially, int_vect[] (subset of the minimum set) is computed by using cv vectors maintained at the initiator MSS ; on receiving response() from some MSS: int_vect=int_vect∪ np_int_vect; after receiving responses from all relevant processes, int_vect[] contains the exact minimum set; '∪', is a operator for bitwise logical OR; np_int_vect is described later;

R[]: a bit vector of length n; R[i]=1 implies P_i has taken its ad hoc checkpoint;

timer1: a flag; initialized to '0' when the timer is set; set to '1' when maximum allowable time for collecting coordinated checkpoint expires;

T[] a bit vector of length n; T[i]=1 implies P_i has taken its tentative checkpoint;

(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 ad hoc checkpoint;

E[]: a bit vector of length n; E[i] is set to '1' if ad hoc checkpoint checkpoint request is sent to P_i and P_i is in the cell;

F[] a bit vector of length n; F[i] is set to '1' if the tentative checkpoint request is sent to P_i;

FF[] a bit vector of length n; FF[i] is set to '1' if P_i is in its cell and it has taken its tentative checkpoint;

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;

p_cn_{in} P_Cnof initiator process;

comm_csn_vect[] an array of length n for n processes; comm_csn_vect[j] denotes the P_j's most recent committed checkpoint's csn; on commit, for all j, (if int_vect [j]==1) comm_csn_vect[j]++; int_vect[] is the exact minimum set received along with the commit request; comm_csn_vect[] is not updated on tentative or ad hoc checkpoints; we maintain one comm_csn_vect array for each MSS and not for each process;

tnp_int_vect a bit vector of length n; it contains the new processes found for the minimum set while executing a potential checkpoint request [Refer Section 4.5.1];

np_int_vect 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_int_vect≠ϕ) np_int_vect= np_int_vect∪ tnp_int_vect

tint_vect a bit vector of length n; tint_vect[k]=1 implies P_k belongs to the minimum set; it maintains the local knowledge of the minimum set; on receiving tint_vect, int_vect, tnp_int_vect along with ad_req (checkpoint request): tint_vect=tint_vect ∪ad_req.tint_vect, tint_vect=tint_vect ∪ad_req.int_vect, tint_vect=tint_vect ∪ad_req.tnp_int_vect; on each potential checkpoint request, tnp_int_vect is computed, if (tnp_int_vect≠ϕ) tint_vect= tint_vect∪ tnp_int_vect ;

chkpt a flag; set to 1 when the MSS learns that some checkpointing process is going on;

ad_req a checkpoint request; when MSS_{in} sends ad hoc checkpoint request (ad_req) to MSS_p, it piggybacks the data structures: P_{in}, MSS_{in}, p_cn_{in}, MSS_p, int_vect; any other MSS piggybacks tint_vect, tnp_int_vect in place of int_vect;

2.3.1 Computation of int_vect or tnp_int_vect:

Let D be the bit dependency matrix of n*n, where j^{th} row denote the cv[] 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 cv[] vector is assumed. Initial cv[] of P_k is: ∀i, cv[i]=0; cv[k]=1.

(a) Computation of int_vect[]: Let P_i be the initiator process.

A= cv_i[]; int_vect=cv_i[]; A=A×D;
While (A≠int_vect[]) do { int_vect=A; A= A×D;}

(b) Computation of tnp_int_vect:
A=tint_vect; B=tint_vect; B=B×D;
While (A≠B) do { A=B; B= B×D;}
Initialize tnp_int_vect;
for(i=0;i<n;i++)
If(A[i]==1∧tint_vect[i]==0) tnp_int_vect[i]=1;

Brief Description of the Algorithm along with an Example

We explain our checkpointing algorithm with the help of an example. In Figure 2, at time t_1, P_2 initiates checkpointing process. $cv_2[1]=1$ due to m_1; and $cv_1[4]=1$ due to m_2. On the receipt of m_0, P_2 does not set cv_2 [3] =1, because, P_3 has taken its 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 int_vect (subset of minimum set) on the basis of cv vectors maintained at MSS_{in}, which in case of Figure 4.2 is {P_1, P_2, P_4}. Therefore, P_2 sends ad hoc checkpoint request to P_1 and P_4 and takes its own ad hoc checkpoint. After taking its ad hoc checkpoint, P_1 sends m_4 to P_4. P_4 logs m_4^{-1} [Refer Section 4.7 and 4.9]. In this case, P_1 has taken its checkpoint before sending m_4; at the time of receiving m_4, P_4 has not taken its checkpoint for the current initiation. If P_4 takes checkpoint after receiving m_4, them m_4 will become orphan. Therefore P_4 logs $m_4^{-1.}$ On recovery, P_4 will receive m_4 as duplicate message because the processes are deterministic and m_4 will be annihilated by m_4^{-1}. Hence receive of m_4 as duplicate message will not cause any inconsistency. It should be noted that this scheme is not applicable for non-deterministic systems. After taking its ad hoc checkpoint C_{41}, P_4 also finds that it was dependent upon P_5 before taking the checkpoint due to m_6 and P_5 is not in the minimum set computed so far. Therefore, P_4 sends ad hoc checkpoint request to P_5. On receiving the checkpoint request, P_5 takes its ad hoc checkpoint. At time t_2, P_2 receives responses from all relevant processes and sends the tentative checkpoint 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 tentative checkpoint request, it converts its ad hoc checkpoint into tentative one. Finally, at time t_3, P_2 sends the commit message to all concerned processes. In this example, {C_{00}, C_{11}, C_{21}, C_{30}, C_{41}, C_{51}, m_4^{-1}} constitute a recovery line. It should be noted that, in the recorded global state, m_4 is an orphan message and its anti-message is also recorded at the receiver end.

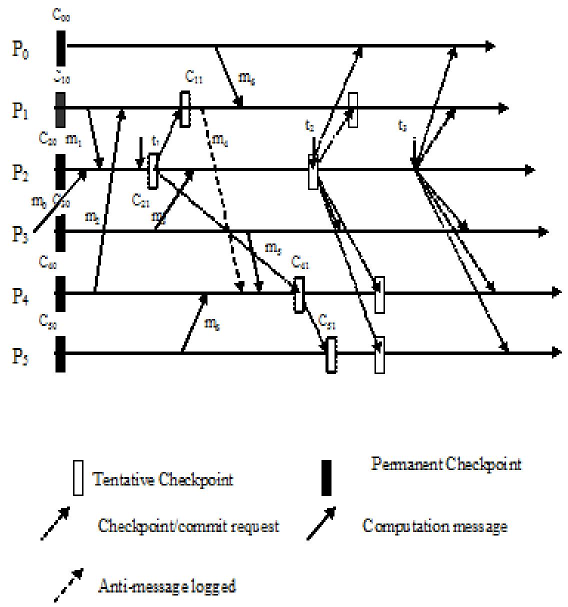

Figure 2

2.3.2 The Proposed Checkpointing Algorithm

When an MH sends an application message, it needs to first send to its local MSS over the wireless cell. The MSS can piggyback appropriate information onto the application message, and then route it to the appropriate destination. Conversely, when the MSS receives an application message to be forwarded to a local MH, it first updates the relevant vectors that it maintains for the MH, strips all piggybacked information from the message, and then forwards it to the MH. Thus, 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 MSS.

Each process P_i can initiate the checkpointing process. Initiator MSS (say MSS_{in}) initiates and coordinates checkpointing process on behalf of MH_i. It computes int_vect (subset of the minimum set on the basis of direct dependencies maintained locally) ; and sends ad hoc checkpoint request (say ad-req) along with int_vect to an MSS if the later supports at least one process in the int_vect. It also updates its tint_vect on the basis of int_vect. We assume that concurrent invocations of the algorithm do not occur.

On receiving the ad-req, along with the int_vect from the initiator MSS, an MSS, say MSS_i, takes the following actions. It updates its tint_vect on the basis of int_vect. It sends the ad_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 int_vect and (iii) ad_req has not been sent to P_i. If no such process is found, MSS_i ignores the ad_req. Otherwise, on the basis of tint_vect, cv vectors of processes in its cell, initial cv vectors of other processes, it computes tnp_int_vect. If tnp_int_vect is not empty, MSS_i sends ad_req along with tint_vect, tnp_int_vect to an MSS, if the later supports at least one process in the tnp_int_vect. MSS_i updates np_int_vect, tint_vect on the basis of tnp_int_vect and initializes tnp_int_vect.

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

For a disconnected MH, that is a member of minimum set, the MSS that has its disconnected checkpoint, converts its disconnected checkpoint into the required one.

When an MSS learns that all of its relevant processes have taken their ad hoc checkpoints successfully or at least one of its processes has failed to take its adhoc checkpoint, it sends the response message along with the np_int_vect to the initiator MSS. If, after sending the response message, an MSS receives the checkpoint request along with the tnp_int_vect, and learns that there is at least one process in tnp_int_vect 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 int_vect on the basis of np_int_vect, received along with the response. Finally, initiator MSS sends tentative checkpoint request to all the processes of the minimum set . In this case, if some process fails to take ad hoc checkpoint in the first phase, then concerned MHs need to abort their ad hoc checkpoints only. The effort of taking an ad hoc checkpoint is insignificant as compared to the tentative one. In this way, the loss of checkpointing effort, in case of an abort of the checkpointing procedure, is significantly low.

When a process in the minimum set receives the tentative checkpoint request, it converts its ad hoc checkpoint into tentative one. In the third phase, 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 *int_vect* [], convert their tentative checkpoints into permanent ones. On receiving commit or abort, all processes update their *dependency* vectors and other data structures.

2.3.3 Handling Node Mobility and Disconnections

Disconnection of an MH is a voluntary operation, and frequent disconnections of MHs is an expected feature of a mobile distributed system. Abrupt disconnections due to battery failure, process failure, or network failure are different from voluntary disconnections [1].
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 [1].
Suppose, an MH, say MH_i, disconnects from the MSS, say MSS_k. MH_i takes its checkpoint, say d_ckpt_i, and transfers it to MSS_k. MSS_k stores all the relevant data structures and d_ckpt_i of MH_i on stable storage. If MH_i is in the int_vect[], d_ckpt_i is considered as MH_i's checkpoint for the current initiation. On commit, MSS_k also updates MH_i's data structures, e.g., cv[], send, etc. On the receipt of messages for MH_i, MSS_k does not update MH_i's *cv*[], but maintains a message queue to store the messages.
When MH_i enters in the cell of MSS_j, it is connected to the MSS_j if no checkpointing process is going on. Before connection, MSS_j collects its cv[], buffered messages, etc. from MSS_k; and MSS_k discards MH_i's support information and d_ckpt_i. The stored messages are processed by MH_i, in the order of their receipt at the MSS. MH_i's cv[] is updated on the processing of buffered messages. If a node does not reconnect in a stipulated time, then its computation can be restarted from its d_ckpt.

3. 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 [1]. 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 [4], [5]. 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. In our scheme, if any process fails to take its ad hoc checkpoint in the first phase, all concerned processes abort their ad hoc checkpoints only; and the loss of checkpointing effort is quite low as compared to other protocols [2, 4, 3, 6], in which every concerned process is forced to abort its tentative checkpoint. In our scheme, if any process fails to convert its ad hoc checkpoint into tentative one, then we propose to follow the technique proposed by Kim & Park [7] in which 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 second phase of checkpointing, total abort of the checkpointing is avoided.

4. PERFORMANCE EVALUATION

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

The Synchronization message overhead:
In the first phase, a process taking an ad hoc checkpoint needs two system messages: request and reply. However, we have used some techniques to reduce the duplicate checkpoint requests. Thus the system overhead is approximately $2*N_{min}*C_{pp}$ in the first phase. Similarly, system overhead in the second phase is: $2*N_{min}*C_{pp.}$ In the first phase we broadcast the adhoc checkpoint request. In the second phase, the tentative requested is broadcasted on the static network; and the system overhead is C_{bst}. In the third phase, we broadcast the commit request. The total message overhead comes out to be: $4*N_{min}*C_{pp}+ 3C_{bst}$
Number of processes taking checkpoints: It requires only minimum number of processes to take their checkpoints.
In minimum-process coordinated checkpointing, some useless checkpoints are taken which are discarded on commit [2, 8, 14]; or some blocking of processes takes place during checkpointing [4, 6, 9]. In the proposed scheme, no useless checkpoints are taken and no blocking of processes takes place. We log anti-messages of very few

messages at the receiver's end only during the checkpointing period. The effort of logging few anti-messages may be negligibly small as compared to taking some useless checkpoints or blocking some processes during checkpointing especially in mobile distributed systems.

The blocking time of the Koo-Toueg [4] protocol is highest, followed by Cao-Singhal [6] algorithm. The other schemes are non-blocking [2, 3, 13], like the proposed one. In Elnozahy et al [3] algorithm, all processes are required to take their checkpoints in an initiation. In the protocols [6], [4], and the proposed one, only minimum numbers of processes record their checkpoints.

Table 1 A Comparison of System Performance

	Cao-Singhal [6]	**Koo-Toeg Algorithm [4]**	**Elnozahy et al [3]**	**Proposed Algorithm**
Avg. blocking Time	$2T_{st}$	$N_{min}*T_{ch}$	0	0
Average No. of checkpoints	N_{min}	N_{min}	N	N_{min}
Average Message Overhead	$3C_{bst}+2C_{wireless}+2N_{mss}*C_{st}+3N_{mh}*C_{wl}$	$3*N_{min}*C_{pp}*N_{dep}$	$2*C_{bst}+N*C_{pp}$	$4*N_{min}*C_{pp}+3C_{bst}$

The message overhead in the proposed protocol is greater than [2, 3, 4, 6] due to the fact that the proposed scheme is a three phase algorithm. Our algorithm is a three phase algorithm; therefore it suffers from extra message overhead of $C_{bst}+2N_{min}*C_{wl}$. By doing so, we are able to reduce the loss of checkpointing effort in case of abort of the checkpointing procedure in the first phase. In other algorithms [2, 3, 4, 6, 13], in case of abort in the first phase, all concerned processes are forced to abort their tentative checkpoint whereas in the proposed scheme, all relevant processes abort their ad hoc checkpoints only. The effort of taking an ad hoc checkpoint is negligible as compared to tentative one in the mobile distributed system [2]. Frequent abort of checkpointing algorithms, due to exhausted battery power, abrupt disconnections etc., may significantly increase the checkpointing overhead in two-phase algorithms. We try to minimize the same by designing the three phase algorithm at the cost of slight increase in message overhead.

The algorithms proposed in [2, 3, 4, 6, 8, 9] assume that the processes are non-deterministic, whereas, we assume in the proposed algorithm that the processes are deterministic in nature as in [13].

5. CONCLUSIONS

In this chapter, we have proposed a minimum-process non-intrusive checkpointing protocol for deterministic mobile distributed systems, where no useless checkpoints are taken and no blocking of processes takes place. In minimum-process checkpointing protocols, some useless checkpoints are taken or blocking of processes takes place; we eliminate both by logging anti-messages of selective messages at the receiver end only during the checkpointing period. The overheads of logging a few anti-messages may be negligible as compared to taking some useless checkpoints or blocking the processes during checkpointing especially in mobile distributed system. We also try to reduce the loss of checkpointing effort when any process fails to take its checkpoint in coordination with others in the first phase. In case of a failure during checkpointing in the first phase, all concerned processes need to abort their ad hoc checkpoints only. The cost of taking an ad hoc checkpoint is negligibly small as compared to the tentative one especially in case of mobile distributed systems. In case, some process fails to convert its ad hoc checkpoint into tentative one, then we follow the selective commit mechanism, in which a process commits its checkpoint if none of the process, it causally depends upon, fails to take its tentative checkpoint. We disallow concurrent executions in spite of concurrent initiations of the proposed protocol.

6. REFERENCES

[1] Acharya A., "Structuring Distributed Algorithms and Services for networks with Mobile Hosts", Ph.D. Thesis, Rutgers University, 1995.

[2] 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.

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

[4] 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.

[5] 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, October 1996.

[6]. G. Cao and M. Singhal. "On impossibility of Min-Process and Non-Blocking Checkpointing and An Efficient Checkpointing algorithm for mobile computing Systems". OSU Technical Report #OSU-CISRC-9/97-TR44, 1997.

[7] 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.

[8] P. Kumar, L. Kumar and R.K. Chauhan, "A Non-Intrusive minimum process synchronous checkpointing protocol for mobile distributed systems", in proceeding of IEEE ICPWC-2005,2005.

[9] Parveen Kumar, "A Low-Cost Hybrid Coordinated Checkpointing Protocol for mobile distributed systems", Mobile Information Systems. pp 13-32, Vol. 4, No. 1, 2007.

[10] David R. Jefferson, "Virtual Time", ACM Transactions on Programming Languages and Systems, Vol. 7, NO.3, pp 404-425, July 1985.

[11] Johnson, D.B., Zwaenepoel, W., " Sender-based message logging", In Proceedingss of 17th international Symposium on Fault-Tolerant Computing, pp 14-19, 1987.

[12] Johnson, D.B., Zwaenepoel, W., "Recovery in Distributed Systems using optimistic message logging and checkpointing. pp 171-181, 1988.

[13] Pushpendra Singh, Gilbert Cabillic, "A Checkpointing Algorithm for Mobile Computing Environment", LNCS, No. 2775, pp 65-74, 2003.

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

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

[16] Sunil Kumar, R K Chauhan, Parveen Kumar, "A Minimum-process Coordinated Checkpointing Protocol for Mobile Computing Systems", *International Journal of Foundations of Computer science*,Vol 19, No. 4, pp 1015-1038 (2008).