International Journal of Computer Applications (0975 – 8887)
Volume 3 – No.1, June 2010

Anti-message Logging based Coordinated Checkpointing Protocol for Deterministic Mobile Computing Systems

Praveen Kumar
Department of Computer Science & Engineering,
Meerut Institute of Engineering & Technology,
Meerut, India

Ajay Khunteta
Singhaniya University
Pechri, Rajasthan, India

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 minimum-process coordinated checkpointing algorithm for deterministic mobile distributed systems, where no useless checkpoints are taken, no blocking of processes takes place, and anti-messages of very few messages are logged during checkpointing. We try to reduce the loss of checkpointing effort when any process fails to take its checkpoint in coordination with others. We also address the related issues like: failures during checkpointing, disconnections, concurrent initiations of the algorithm.

1. INTRODUCTION

1.1 Definitions and Notations

Checkpoint: Checkpoint is defined as a designated place in a program at which normal process is interrupted specifically to preserve the status information necessary to allow resumption of processing at a later time. A checkpoint is a local state of a process saved on stable storage. By periodically invoking the checkpointing process, one can save the status of a program at regular intervals.

Rollback Recovery: If there is a failure one may restart computation from the last checkpoints thereby avoiding repeating computation from the beginning. The process of resuming computation by rolling back to a saved state is called rollback recovery.

Global State : 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.

Orphan Message: A message whose receive event is recorded, but its send event is lost.

Consistent Global State: A global state is said to be “consistent” if it contains no orphan message. 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. In distributed systems, checkpointing can be independent, coordinated [3], [8], [11] or quasi-synchronous [2], [9]. Message Logging is also used for fault tolerance in distributed systems [14].

Asynchronous Checkpointing: Under the asynchronous approach, checkpoints at each process are taken independently without any synchronization among the processes. Because of absence of synchronization, there is no guarantee that a set of local checkpoints taken will be a consistent set of checkpoints. It may require cascaded rollbacks that may lead to the initial state due to domino-effect [7].

Coordinated Checkpointing: 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], [8], [11], [22]. 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. Communication-induced Checkpointing: It avoids the domino-effect without requiring all checkpoints to be coordinated [2], [7], [9]. In these protocols, processes take two kinds of checkpoints, local and forced. Local checkpoints can be taken independently, while forced checkpoints are taken to guarantee the eventual progress of the recovery line and to minimize useless checkpoints. As opposed to coordinated checkpointing, these protocols do no exchange any special coordination messages to determine when forced checkpoints should be taken. But, they piggyback protocol specific information [generally checkpoint sequence numbers] on each application message; the receiver then uses this information to decide if it should take a forced checkpoint.

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 [23, [24], [25].

Checkpoint Interval (CI): The i^{th} CI of a process denotes all the computation performed between its i^{th} and $(i+1)^{th}$ checkpoint, including the i^{th} checkpoint but not the $(i+1)^{th}$ checkpoint.

Direct Dependency among Processes: 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 CI and P_k has not taken its permanent checkpoint after sending m.

Minimum Set: A process P_i is in the minimum set only if checkpoint initiator process is transitively dependent upon it.

Minimum-process Coordinated Checkpointing Algorithms : In these algorithms, only a subset of interacting processes (called minimum set) are required to take checkpoints in an initiation.

Anti-Message: David R. Jefferson [29] 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}.

1.2 Proposed Work

In the present study, 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 when any process fails to take its checkpoint.

2. THE PROPOSED CHECKPOINTING ALGORITHM

2.1 System Model

Our system model consists of a number of MHs which communicate through mobility support stations (MSSs). Each MSS is a fixed network host which provides wireless communication support for a fixed geographical area, called a cell. MSSs are linked together over the wired data networks. The distributed system consisting of n processes, running on MHs or MSSs. The MHs can communicate with the MSS through wireless channels. We assume that wireless channels and logical channels are all FIFO order. If a MH moves to the cell of another base station, a wireless channel to the old MSS is disconnected and a wireless channel in the new MSS is allocated. However, its checkpoint related information is still with the old MSS. A MH may voluntarily disconnect from mobile computing networks. The MH does not send and receive any message when it is in a disconnected state. We also assume a closed system that consists of nodes, links, and disks. Input is stored on disk before operation begins. Output is stored on disk when the job ends.

There is no common clock, shared memory or central coordinator. Message passing is the only mode of communication between any pair of processes. The messages originated from a source Mh, are received by the local Mobile support stations and then forwarded to the destination MH. Any process can initiate checkpointing. It is assumed that processes may be failed during processing but there is no communication link failure. Messages are exchanged with finite but arbitrary delays. In our algorithm, we consider that the processes which are running in the distributed mobile systems are deterministic.

2.2 Basic Idea

In coordinated checkpointing for mobile distributed systems, there remains a good probability that some process fails to take its checkpoint especially on MH. It can happen due to abrupt disconnection, exhausted battery power, or failure in wireless bandwidth. If a single process fails to take its checkpoint; all the checkpointing effort goes waste, because, each process has to abort its tentative checkpoint. 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 quite high. Therefore, we propose that in the first phase, all concerned MHs will take mutable checkpoint only. Mutable checkpoint is described in [5], it is stored on the memory of MH only. In this case, if some process fails to take its checkpoint in the first phase, then other MHs need to abort their mutable checkpoints only. The effort of taking a mutable checkpoint is negligible as compared to the tentative one [5]. When the initiator process comes to know that all relevant processes have taken their mutable checkpoints, it asks all relevant processes to come into the second phase, in which, a process converts its mutable checkpoint into tentative one, i.e. an MH transfers its checkpoint to its local MSS over wireless channel.

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. Suppose, P_i processes m, and it receives checkpoint request later on, and then it takes its checkpoint. In that case, m will become orphan in the recorded global state. 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 or blocking of processes takes place. The overheads of logging a few anti-messages may be negligible as compared to taking some useless checkpoints or blocking the processes during checkpointing. This scheme will not be applicable for non-deterministic systems. For non-deterministic systems, we can not deal with an orphan message by logging its anti-message at the receiver end; because, in non-deterministic systems, if we log m_1^{-1} of orphan message m_1, then after recovery, we may get m_2 in place of m_1, which may be different from m_1. In that case, m_1^{-1} can not annihilate m_2, and

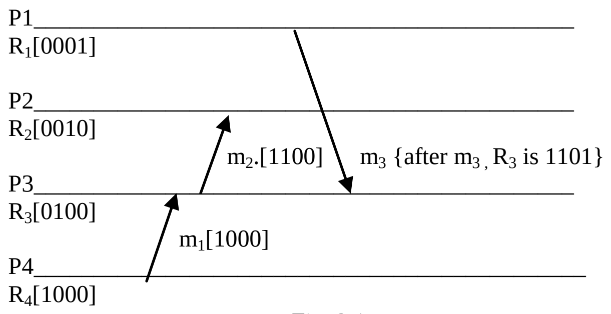

Fig. 2.1

hence the inconsistency will take place. If P_i become sure that it is not good to take its checkpoint for current initiation or, if P_i has already taken its checkpoint for the current initiation then it can process m without any issue. In this case m can not become orphan.

In the figure 2.1 P_4 sends m_1 to P_3 along with its own dependency vector R_4[1000]. When P_3 receives m_1 it update its own dependency vector by taking logical OR of R_4 & R_3[0100], which comes out to be 1100. When P_3 send m_2 to P_2, it appends R3[1100] along with m_2. When P_2 receive m_2, it updates its own dependency vector R_2 by taking logical OR of R_2 and R_3, which comes out to be [1110]. In this way, partial transitive dependencies are captured during normal computation. It should be noted that all the transitive dependencies are not captured during normal computation. At time t1, the dependency vector of P_2 shows that P_2 is not transitively dependent upon P_1, whereas, P_2 is transitively dependent upon P_1due to m_3 and m_2.

2.3 An Example

Example: we explain the purposed algorithm with the help of an example. In figure 2.2, at time t_1, P_3 initiates checkpointing. At this moment, it dependency vector is[000111]. P_1 sends m_1 along with its dependency vector [000001]. When P_2 receives m it update its dependency vector by taking bitwise logical OR of dependency vectors of P_1 and P_2, which comes out to be [000011]. On receiving m2 the dependency vector of P3 become [000111]. At time t_1, the P_3 computes tentative minimum set{P_1, P_2, P_3}and sends the mutable checkpoint request w P1 & P2 and takes its own mutable checkpoint. For an MH the mutable checkpoint is stored on the disk of MH. After taking its mutable checkpoint, P_3 sends m_4 to P_4. P_3 also piggyback its own csn along with m_4. When P_4 receive m_4, it finds that csn of P_3 at the time of sending m_4 is 1; therefore, P_4 concludes that P_3 has taken its checkpoint for the new initiation and P_4 has not taken checkpoint for the same. Therefore, P_4 logs m_4^{-1} and process m_4. when P_2 takes its mutable checkpoint C_{21}, it finds that it is dependent upon P_4, due to m_3 and P_4 is not in the tentative minimum set {P_1,P_2,P_3},

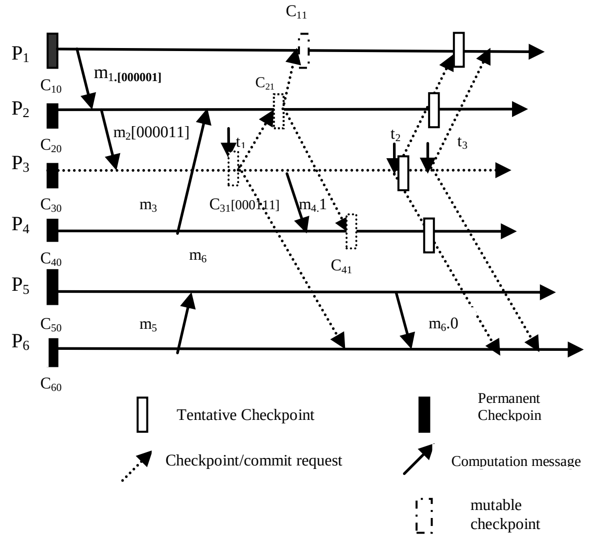

Figure 2.2

therefore, P_2 sends checkpoint request to P_4 and P_4 takes its mutable checkpoint C_{41}. P_5 sends m_6 to P_6. on the basis of piggybacked csn of P_5, on m_6, P_6 concludes that P_5 has not taken it checkpoint for the current initiation, therefore P6 process m6 without logging m_6^{-1}. At time t_2, P_3 receives response form all process in the minimum set(not shown in the figute) and finds that they have taken their mutable checkpoints successfully, therefore P_3 issues tentative checkpoint request to all processes. On getting tentative checkpoint request, a process converts it mutable checkpoint into tentative one and sends the response to initiator process P_3. At time t_3, P_3 receives responses form the process in minimum set and finds that they have taken their tentative checkpoints successfully, therefore, P_3 issues commit request to all process. A process in the minimum set converts its tentative checkpoint into permanent checkpoint and discards it old permanent checkpoint if any. The anti messages are also logged on stable storage. In the present case the global checkpoint is collected as follows:{C_{11}, C_{21}, C_{31}, C_{41}, m_4^{-1}, C_{50}, C_{60}). If the system recovers from this state after a fault, P_3 will send m_4 again as the processes are assumed to be deterministic. When P_4 will receive m_4, it will find that m_4^{-1} is logged at P_4. P_4 concludes that it has already received m_4, therefore P_4 ignores m_4.

2.4 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.

Pr_csn_i : A monotonically increasing integer checkpoint sequence number for each process. It is incremented by 1 on mutable checkpoint.

td_vect_i []: It is a bit array of length n for n process in the system. td_vect_i[] =1 implies P_i is transitively dependent upon P_j. When P_i receives m from P_j such that P_j has not taken any permanent checkpoint after sending m then P_i sets $td\_vect_i[j]=1$. When P_i commit its checkpoint, it sets $td\_vect_i[]=0$ for all processes except for itself which is initialized to 1.

chkpt-st_i: A boolean which is set to ‘1’ when P_i takes a tentative checkpoint; on commit or abort, it is reset to zero

m_vect[]: An bit array of size n for n processes in the systems. When P_i starts checkpointing procedures, it computes tentative minimum set as follows: m_vect[j] = $td\_vect_i[j]$ where j=1,2, ….,n.

TC[]: An array of size n to save information about the processes which have taken their tentative checkpoints in the second phase. When process P_j takes its tentative checkpoint then j^{th} bit of this vector is set to 1. It is initialized to all zeros in the beginning of the checkpointing process. It is maintained by the checkpoint initiator MSS only.

MC[]: A bit array of size n, maintained by initiator MSS. MC[i]=1 implies P_i has taken its mutable checkpoint in the first phase.

MSS_chk_taken2[]: A bit array of length n maintained by each MSS. MSS_chk_taken2[i] =1 implies P_i has taken its tantative checkpoint successfully in the second phase.

MSS_chk_request2[]: A bit array of length n at each MSS. MSS_chk_request2[i] =1, Pi has been issued tentative checkpoint request in the second phase.

Max_time: it is a flag used to provide timing in checkpointing operation. It is initialized to zero when timer is set and becomes ‘1’ when maximum allowable time for collecting global checkpoint expires.

MSS_plist[]: A bit array of length n for n processes which is maintained at each MSS $MSS\_plist_K[j]=1$ implies each process P_j is running on MSS_k. If P_j is disconnected, then it checkpoint related information is on MSS_k.

MSS_chk_taken: A bit array of length n bits maintained by the MSS. MSS_chk_taken [j]=1 implies P_j which is in the cell of MSS has taken its mutable checkpoint in the first phase.

MSS_chk_request: A bit array of length n at each MSS. The j^{th} bit of this array is set to ‘1’ whenever initiator sends the checkpoint request to P_j and P_j is in the cell of this MSS.

MSS_fail_bit: A flag maintained on every MSS, initialized to ‘0’; set to ‘1’ when any process in the cell of MSS fails to take tentative checkpoint
P_{in} : The process which has initiated the checkpointing operation
MSS_{in} : The MSS which has P_{in} in its cell
p_csn_{in} : checkpoint sequence number of initiator process
g_chkpt: A flag which indicates that some global checkpoint is being saved
csn[]: An array of size n, maintained on every MSS, for n processes. csn[i] represens the most recently committed checkpoint sequence number of P_i. After the commit operation, if m_vect[i]=1 then csn[i] is incremented. It should be noted that entries in this array are updated only after converting tentative checkpoints in to permanent checkpoints and not after taking tentative checkpoints.
m_vect1[]: An array of size n maintained on every MSS. It contains those new processes which are found on getting checkpoint request from initiator.

m_vect2[]: An array of size n. for all j such that m_vect1[j] $\neq$0,

m_vect2= m_vect2$\cup$ m_vect1.
m_vect3[]: An array of length n; on receiving m_vect3[], m_vect[], m_vect1[] along with checkpoint request [c_req] or on the computation of m_vect1[] locally: m_vect3[]=m_vect3[] $\cup$ c_req.m_vect3[]; m_vect3[]=m_vect3[]$\cup$m_vect[]; m_vect3[]=m_vect3[] $\cup$ c_req.m_vect1[]; m_vect3[]=m_vect3[] $\cup$ m_vect1[]; m_vect3[] maintains the best local knowledge of the minimum set at an MSS;

2.5 Computation of m_vect[], m_vect1[], m_vect2[], m_vect3[]:

1. Suppose a process P_r wants to initiate checkpointing procedure. Its send its request to its local MSS, say $MSS_{r..}$ MSS_r maintains the dependency vector of P_r (say td_vect_r[]).MSS_r coordinates checkpointing on behalf of $P_{r.}$ It computes tentative minimum set as follows:

$\forall_{i=1,n}$ m_vect[i] = td_vect_r[i]

2. On receving m_vect[] from $MSS_{r,}$ any MSS (say MSS_S) computes the m_vect1[] as follows:

Suppose MSSs maintains the process P_j such that $P_j \in$ MSSs and

$P_j \in$ m_vect

$\forall i,$ m_vect1[i]=1 iff m_vect[i]=0 and td_vect_j[i]=1
m_vect1[] maintains the new processes found for the minimum set when a process receives the checkpoint request.
m_vect2=m_vect2 U m_vect1
$\forall$ i, m_vect1[i]=0

3. m_vect3= m_vect U m_vect2
MSS_{in} sends c_req to MSS_s along with m_vect[]and some process (say P_k) is found at MSS_s, which takes the checkpoint to this c_req. All MSSs maintains the processes of minimum set to the best of their knowledge in m_vect3. 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 m_vect3 , then MSS_s sends 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 m_vect1. When an MSS finds that all the local processes, which were asked to take checkpoints, have taken their checkpoints, it sends the response to the MSS_{in} along with m_vect2; so that MSS_{in} may update its knowledge about minimum set and wait for the new processes before sending commit. In this way, MSS_{in} sends commit only if all the processes in the minimum set have taken their tentative checkpoints.

2.6. The Checkpointing Protocol

As the wireless bandwidth is a scarce commodity in mobile systems; therefore; we impose minimum burdon on wireless channels. The local MSS of an MH acts on behalf of the process running on MH. We piggyback checkpoint sequence numbers and dependency vectors onto normal computation messages, but this information is not sent on wireless channels. The local MSS of an MH, strips all the additional information from the computation message and sends it to the concerned MH. The dependency vector of a process running on an MH is maintained by its local MSS.
Our algorithm is distributed in nature in the sense that any process can initiate checkpointing. If two processes initiate checkpointing concurrently, then the checkpoint imitator of the lower process ID will prevail. The local MSS of a process coordinates checkpointing on its behalf. Suppose two processes P_i and P_j starts checkpointing concurrently and MSS_p and MSS_q are their local MSS respectively then MSS_p and MSS_q will send checkpoint requests along with tentative minimum set to all the MSS’s. MSSp will receive the checkpoint request of MMSq and MMSq will receive the checkpoint request of MSSp. Suppose Process-ID of P_i is less than Process-ID of P_j, then the checkpoint initiates of P_i will prevail. Any other MSS will automatically ignore the request of P_j because, every MSS will compare the process id of P_i and P_j.
We propose that any process in the system can initiate the checkpointing operation. When a process P_{in} starts checkpointing procedure, it send its request to its local MSS say MSS_{in}. MSS_{in} computes the tentative minimum set (m_vect[]) as follows:

$\forall_{i=1,n}$ m_vect[i] = td_vect[i]. MSS_{in} coordinates checkpointing process on behalf of $P_{in.}$ We want to emphasize that td_vect_{in}[] contains the processes on which P_{in} transitively depends and the set is not complete.
MSS_{in} sends c-req (mutable checkpoint request) to all MSS’s along with m_vect_{in}[]. When an MSS say MSS_p receives c-req; it sends the c-req to all such process which are running in its cell and are also the member of m_vect_{in}[]. Suppose, P_j gets the checkpoint request at $MSS_{p.}$ Now we find any process P_k such that P_k does not belong to m_vect_{in}[] and P_k belongs to td_vect_j[]. In this case, P_k is also included in the minimum set. During checkpointing, suppose P_i takes it checkpoint and after that it send m to P_j such that P_j has not taken its checkpoint at the time of receiving m. If P_j receive m and it gets checkpoint request later on then m will become orphan. In order to handle this situation, we log m^{-1} at P_j. if P_j takes the checkpoint, then m becomes the orphan message in the recorded global state. In deterministic system, orphan message will be delivered again at receiving and it will be annihilated by its anti-message stored at the receiver end. Suppose, P_i sends m to P_j and P_j has already logged m^{-1}. it implies P_j has already received m and m is a duplicate message, therefore, P_j ignores m and also delete m^{-1}.
For a disconnected MH that is a member of minimum set, the MSS that has its disconnected checkpoint, considers its disconnected checkpoint as the required come. When a MSS learns that its concerned processes in its cell have taken their mutable checkpoints, it sends the response to MSS_{in} . It should be noted that in the first phase, all processes take the mutable

checkpoints. For a process running on a static host, mutable checkpoint is equivalent to tentative checkpoint. But, for an MH, mutable checkpoint is different from tentative checkpoint. In order to take a tentative checkpoint, an MH has to record its local state and has to transfer it to its local MSS. But, the mutable checkpoint is stored on the local disk of the MH. It should be noted that the effort of taking a mutable checkpoint is very small as compared to the tentative one[5]. When the initiator MSS comes to know that all processes in the minimum set taken their mutable checkpoint successfully MSS_{in} issues tentative checkpoint request to all MSSs along with the exact minimum set. Alternatively, if MSS_{in} comes to know that some process has failed to take its checkpoint in the first phase, it issues abort request to all MSS. In this way the MHs need to abort only the mutable checkpoints, and not the tentative ones. In this way we try to reduce the loss of checkpointing effort in case of abort of checkpointing algorithm in first phase.

When an MSSs receives the tentative checkpoint request, it asks all relevant process to convert their mutable checkpoints into tentative ones. In order to convert its mutable checkpoint into tentative one, an MH needs to transfer its checkpoint data to its local MSS. A process on static node has to do nothing in order to convert its mutable checkpoint into tentative one. When an MSS learns that all its relevant processes have taken their tentative checkpoint, it informs $MSS_{in.}$ Finally, when the initiator MSS learns that all processes in the minimum set have taken their tentative checkpoints successfully, it issues commit request to all MSSs. When a process in the minimum set gets the commit request, it converts it tentative checkpoint into permanent one and discards its earlier permanent checkpoint, if any. All the anti-messages are also stored on stable storage along with committed checkpoints.

3. PERFORMANCE EVALUATION

In [13], initiator process/MSS collects dependency vectors for all the processes and computes the minimum set and sends the checkpointing request to all the processes with minimum set. The algorithm is non-blocking; the message received during checkpointing may add processes to the minimum set. It suffers from additional message overhead of sending request to all processes to send their dependency vectors and all processes send dependency vectors to the initiator process. But in our algorithm, no such overhead is imposed. The Cao-Singhal [5] suffers from the formation of checkpointing tree as shown in basic idea. In our algorithm, theoretically, we can say that the length of the checkpointing tree will be considerably low as compared to algorithm [5], as most of the transitive dependencies are captured during the normal processing. We do not compare our algorithm with Prakash-Singhal [15], as Cao-Singhal proved that there no such algorithm exists [4].

Furthermore, in [5] algorithm, transitive dependencies are captured by direct dependencies. Hence the average number of useless checkpoints requests will be significantly higher than the proposed algorithm. In [5], huge data structure are piggybacked along with checkpointing request, because they are unable to maintain exact dependencies among processes. Incorrect dependencies are solved by these huge data structures. In our case, no such data structures are piggybacked on checkpointing request and no such useless checkpoint requests are sent., because we are able to maintain exact dependencies among processes and furthermore, are able to capture transitive dependencies during normal computation at the cost of piggybacking bit vector of length n for n processes.

In Cao-Singhal algorithm[5], some useless checkpoints are taken or some blocking of processes takes place, we avoid both by logging anti-messages of very few message at the receiver end only during the checkpoint process. The drawback of our algorithm is that it is not applicable for non deterministic systems while the CS[5] algorithm is designed for non deterministic systems. Our algorithm is distributed in nature that any process can initiate checkpointing. We do not allow concurrent executions of the protocol. If we allow, concurrent executions then our goal of minimizing checkpointing efforts will be defeated, many processes will start taking checkpoint quite frequently without advancing their recovery line significantly.

4. CONCLUSION

In this paper, we have proposed a minimum-process non-intrusive checkpointing protocol for deterministic mobile distributed systems, where no useless checkpoints are taken. The number of processes that take checkpoints is minimized to 1) avoid awakening of MHs in doze mode of operation, 2) minimize thrashing of MHs with checkpointing activity, 3) save limited battery life of MHs and low bandwidth of wireless channels. In minimum-process checkpointing protocols, some useless checkpoints are taken or blocking of processes takes place; we eliminate both by logging anti-messages of very few 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. We try to reduce the checkpointing time by avoiding checkpointing tree which may be formed in Cao-Singhal [5] algorithm. We captured the transitive dependencies during the normal execution by piggybacking dependency vectors onto computation messages. The Z-dependencies are well taken care of in this protocol. We also avoided collecting dependency vectors of all processes to find the minimum set as in [4], [13].

6. REFERENCES

[1]. Acharya A. and Badrinath B. R., “Checkpointing Distributed Applications on Mobile Computers,” Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems, pp. 73-80, September 1994.

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

[3]. Cao G. and Singhal M., “On coordinated checkpointing in Distributed Systems”, IEEE Transactions on Parallel and Distributed Systems, vol. 9, no.12, pp. 1213-1225, Dec 1998.

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

[5]. Cao G. and Singhal M., “Mutable Checkpoints: A New Checkpointing Approach for Mobile Computing systems,” IEEE Transaction On Parallel and Distributed Systems, vol. 12, no. 2, pp. 157-172, February 2001.

[6]. Chandy K. M. and Lamport L., "Distributed Snapshots: Determining Global State of Distributed Systems," ACM Transaction on Computing Systems, vol. 3, No. 1, pp. 63-75, February 1985.

[7]. Elnozahy E.N., Alvisi L., Wang Y.M. and Johnson D.B., "A Survey of Rollback-Recovery Protocols in Message-Passing Systems," ACM Computing Surveys, vol. 34, no. 3, pp. 375-408, 2002.

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

[9]. Hélary J. M., Mostefaoui A. and Raynal M., "Communication-Induced Determination of Consistent Snapshots," Proceedings of the 28th International Symposium on Fault-Tolerant Computing, pp. 208-217, June 1998.

[10]. Higaki H. and Takizawa M., "Checkpoint-recovery Protocol for Reliable Mobile Systems," Trans. of Information processing Japan, vol. 40, no.1, pp. 236-244, Jan. 1999.

[11]. Koo R. and Toueg S., "Checkpointing and Roll-Back Recovery for Distributed Systems," IEEE Trans. on Software Engineering, vol. 13, no. 1, pp. 23-31, January 1987.

[12]. Neves N. and Fuchs W. K., "Adaptive Recovery for Mobile Environments," Communications of the ACM, vol. 40, no. 1, pp. 68-74, January 1997.

[13] Parveen Kumar, Lalit Kumar, R K Chauhan, V K Gupta "A Non-Intrusive Minimum Process Synchronous Checkpointing Protocol for Mobile Distributed Systems" Proceedings of IEEE ICPWC-2005, pp 491-95, January 2005.

[14]. Pradhan D.K., Krishana P.P. and Vaidya N.H., "Recovery in Mobile Wireless Environment: Design and Trade-off Analysis," Proceedings 26th International Symposium on Fault-Tolerant Computing, pp. 16-25, 1996.

[15]. Prakash R. and Singhal M., "Low-Cost Checkpointing and Failure Recovery in Mobile Computing Systems," IEEE Transaction On Parallel and Distributed Systems, vol. 7, no. 10, pp. 1035-1048, October1996.

[16]. Ssu K.F., Yao B., Fuchs W.K. and Neves N. F., "Adaptive Checkpointing with Storage Management for Mobile Environments," IEEE Transactions on Reliability, vol. 48, no. 4, pp. 315-324, December 1999.

[17]. 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.

[18]. L. Kumar, M. Misra, R.C. Joshi, "Checkpointing in Distributed Computing Systems" Book Chapter "Concurrency in Dependable Computing", pp. 273-92, 2002.

[19]. 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.

[20]. Ni, W., S. Vrbsky and S. Ray, "Pitfalls in Distributed Nonblocking Checkpointing", Journal of Interconnection Networks, Vol. 1 No. 5, pp. 47-78, March 2004.

[21]. L. Lamport, "Time, clocks and ordering of events in a distributed system" Comm. ACM, vol.21, no.7, pp. 558-565, July 1978.

[22]. Silva, L.M. and J.G. Silva, "Global checkpointing for distributed programs", Proc. 11th symp. Reliable Distributed Systems, pp. 155-62, Oct. 1992.

[23]. 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.

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

[25]. Lalit Kumar Awasthi, P.Kumar, "A Synchronous Checkpointing Protocol for Mobile Distributed Systems: Probabilistic Approach" International Journal of Information and Computer Security, Vol.1, No.3 pp 298-314.

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

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

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

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

[30] 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).