A NON-BLOCKING MINIMUM-PROCESS CHECKPOINTING PROTOCOL FOR DETERMINISTIC MOBILE COMPUTING SYSTEMS

[1]**Ajay Khunteta,** [2]**Praveen Kumar**

[1],Singhania University, Pacheri, Rajasthan, India-313001

Email: ajay_khunteta@rediffmail.com

[2]Department of Computer Science & Engineering
Meerut Institute of Engineering & Technology, Meerut, India, Pin-250005

ABSTRACT

The term Distributed Systems is used to describe a system with the following characteristics: i) it consists of several computers that do not share memory or a clock, ii) the computers communicate with each other by exchanging messages over a communication network, iii) each computer has its own memory and runs its own operating system. In the mobile distributed system, some of the processes are running on mobile hosts (MHs).A checkpoint algorithm for mobile computing systems needs to handle many new issues like: mobility, low bandwidth of wireless channels, and 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. 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.

Keywords: Checkpointing algorithm, Mobile computing, Distributed Mobile systems etc

1. INTRODUCTION

A distributed system is one that runs on a collection of machines that do not have shared memory, yet looks to its users like a single computer [1]. A distributed system consists of a finite set of processes and a finite set of channels. It can be described by a labeled, directed graph in which the vertices represent processes and the edges represent channels. A computer in distributed system is having two types of resources: i) local resources that are owned and controlled by it, ii) remote resources that are only accessible through network and incurring CPU delay and delay due to communication protocol [1].

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.

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. 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 message whose receive event is recorded, but its send event is lost. 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].

In coordinated or synchronous checkpointing, processes coordinate their local checkpointing actions such that the set of all recent checkpoints in the system is guaranteed to be consistent [6, 8]. In case of a fault, every process restarts from its most recent permanent/committed checkpoint. Hence, this approach simplifies recovery and it does not suffer from domino-effect. Furthermore, coordinated checkpointing requires each process to maintain only one permanent checkpoint on stable storage, reducing storage overhead and eliminating the need for garbage collection. Its main disadvantage is the large latency involved in output commit.

A straightforward approach to coordinate checkpointing is to block communications while the checkpointing process executes. A coordinator takes a checkpoint and broadcasts a request message to all processes, asking them to take a checkpoint. When a process receives a message, it stops its execution, flushes all the communication channels, takes a tentative checkpoint, and sends an acknowledgement message back to the coordinator. After the coordinator receives acknowledgement from all processes, it broadcasts a commit message that completes the two phase checkpointing protocol. After receiving the commit message, each process receives the old permanent checkpoint and makes the tentative checkpoint permanent. The process is then free to resume execution and exchange messages with other processes. The coordinated checkpointing algorithms can also be classified into following two categories: minimum-process and all process algorithms. In all-process coordinated checkpointing algorithms, every process is required to take its checkpoint in an initiation [6], [8]. In minimum-process algorithms, minimum interacting processes are required to take their checkpoints in an initiation.

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]. 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. 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.

A process P_i is in the minimum set only if checkpoint initiator process is transitively dependent upon it. In these algorithms**,** only a subset of interacting processes (called minimum set) are required to take checkpoints in an initiation.

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}.

The Chandy-Lamport [6] algorithm is the earliest non-blocking all-process coordinated checkpointing algorithm. In this algorithm, *markers* are sent along all channels in the network which leads to a message complexity of $O(N^2)$, and requires channels to be FIFO. Elnozahy et al. [8] proposed an all-process non-blocking synchronous checkpointing algorithm with a message complexity of O(N). In coordinated checkpointing protocols, we may require piggybacking of integer csn (checkpoint sequence number) on normal messages [5], [8], [13], [19], [22]. Kumar et al. [18] proposed an all-process non-intrusive checkpointing protocol for distributed systems, where just one bit is piggybacked on normal messages. It results in extra overhead of vector transfers during Checkpointing.

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

The existence of mobile nodes in a distributed system introduces new issues that need proper handling while designing a checkpointing algorithm for such systems. These issues are mobility, disconnection, finite power source, vulnerable to physical damage, lack of stable storage etc. These issues make traditional checkpointing techniques unsuitable to checkpoint mobile distributed systems [1], [5], [15]. To take a checkpoint, an MH has to transfer a large amount of checkpoint data to its local MSS over the wireless network. Since the wireless network has low bandwidth and MHs have low computation power, all-process checkpointing will waste the scarce resources of the mobile system on every checkpoint. Prakash and Singhal [15] gave minimum-process coordinated checkpointing protocol for mobile distributed systems.

A good checkpointing protocol for mobile distributed systems should have low overheads on MHs and wireless channels and should avoid awakening of MHs in doze mode operation. The disconnection of MHs should not lead to infinite wait state. The algorithm should be non-intrusive and should force minimum number of processes to take their local checkpoints [15]. In minimum-process coordinated checkpointing algorithms, some blocking of the processes takes place [4], [11], or some useless checkpoints are taken [5], [13], [19].

Acharya and Badrinath [1] gave a checkpointing protocol for mobile systems. In this approach, an MH takes a local checkpoint whenever a message receipt is preceded by the message sent at that MH. This algorithm has no control over checkpointing activity on MHs and depends totally on communication patterns. In worst case, the number of local checkpoints taken will be equal to the number of computation messages, which may lead to high checkpointing overhead.

Cao and Singhal [5] achieved non-intrusiveness in the minimum-process algorithm by introducing the concept of mutable checkpoints. The number of useless checkpoints in [5] may be exceedingly high in some situations [19]. Kumar et. al [19] and Kumar et. al [13] reduced the height of the checkpointing tree and the number of useless checkpoints by keeping non-intrusiveness intact, at the extra cost of maintaining and collecting dependency vectors, computing the minimum set and broadcasting the same on the static network along with the checkpoint request.

Koo and Toeg [11], and Cao and Singhal [4] proposed minimum-process blocking coordinated checkpointing algorithms. Neves et al. [12] gave a loosely synchronized coordinated protocol that removes the overhead of synchronization. Higaki and Takizawa [10] proposed a hybrid checkpointing protocol where the mobile stations take checkpoints asynchronously and fixed ones synchronously. Kumar and Kumar [29] proposed a minimum-process coordinated checkpointing algorithm where the number of useless checkpoints and blocking are reduced by using a probabilistic approach. A process takes its mutable checkpoint only if the probability that it will get the checkpoint request in the current initiation is high. To balance the checkpointing overhead and the loss of computation on recovery, P Kumar [27] and Kumar et al [26], proposed a hybrid-coordinated checkpointing protocol for mobile distributed systems, where an all-process checkpoint is taken after executing minimum-process checkpointing algorithm for a certain number of times.

Transferring the checkpoint of an MH to its local MSS may have a large overhead in terms of battery consumption and channel utilization. To reduce such an overhead, an incremental checkpointing technique could be used [16]. Only the information, which changed since last checkpoint, is transferred to the MSS.

Johnson and Zwaenepoel [26] 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 [27] used optimistic message logging and checkpointing to determine the maximum recoverable state, where every received message is logged.

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 is similar to [28]. There are *n* spatially separated sequential processes $P_0, P_1,.., P_{n-1}$, 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. A process in the cell of MSS means the process is either running on the MSS or on an MH supported by it. It also includes the processes of MHs, which have been disconnected from the MSS but their checkpoint related information is still with this MSS. We also assume that the processes are deterministic. The i^{th} CI (checkpointing interval) 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.

When an MH sends an application message, it is first sent to its local MSS over the wireless cell. The MSS piggybacks appropriate information with the application message, and then routes it to the destination MSS or MH. When the MSS receives an application message to be forwarded to a local MH, it first updates the data structures that it maintains for the MH, strips all the piggybacked information, and then forwards the message to the MH. Thus, an MH sends and receives application messages that do not contain any additional information; it is only responsible for checkpointing its local state appropriately and transferring it to the local MSS.

2.2 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. $td\_vect_i[j]$ =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.

$chkpt\text{-}st_i$: 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] $\neq$ o, 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;

2.3 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.

2.4 FORMAL OUTLINE OF THE CHECKPOINTING ALGORITHM:

2.4.1 Actions taken when P_i sends m to P_j:

send (P_i,P_j, m, pr_csn$_i$,td_vect$_i$[]);
//P_i piggybacks its own csn and transitive dependency vector onto m.

2.4.2 Algorithm executed at initiator MSS (say MSS_{in})

Suppose P_{in} initiates checkpointing. P_{in} sends the request to MSS_{in}. MSS_{in} computes m_vect [].

```
1. On the basis of computed m_vect, $MSS_{in}$ computes m_vect1, m_vect2, m_vect3 [Refer section 4.1].
2. m_vect = m_vect3.
3. $MSS_{in}$ sends c_req to all MSS  along with m_vect[]//Multiple checkpoint request
4. Set max-time.
5. Wait for response.
6. On receiving response ($P_{in}$, $MSS_{in}$, $MSS_s$, mss_,chk_taken, m_vect2, mss_fail_bit) or at max_time
    6.1 If (max_time)OR(mss_fail_bit) { send message abort (Pin, MSSin, pr_csnin} to all MSSs, Exit;
//Maximum allocated time expired or some process
//failed to take checkpoint
    6.2 m_vect[] = m_vect[] $\cup$ m_vect2[]. ["$\cup$" is a set union operator]
    6.3 MC[] =MC[] $\cup$ mss_chk_taken[]
7. For (k=0;k<n; k++)
 If ( $\exists$ k such that MC[k] $\neq$ m_vect[k]) then go to step 5;
8. S  end message tent_req (Pin, MSSin  , pr_csnin, m_vect[]) to all MSSs;
        // m_vect[] is the exact minimum set//tent_req is tentative checkpoint request.
9. 0n receving response (Pin,MSSin, MSSs, mss_chkpt_taken2[], mss_fail_bit)or at max time
            i) if( (max_time)or (MSS_fail_bit)) send message abort to all MSSs
            ii) TC[]=TC[] U mss_chk_taken2[]
10.         for (k=0; k<n; k++)
```

If (there exit k such that TC[k] not equal m_vect[k] then go to step 9
11. send message commit() to all MSSs.

2.4.3 Algorithm Executed at a process Pj on receiving of m from Pi:

```
Case 1. If (m.pr_csni = = csn[i])// Pi has not taken its checkpoint
// before sending m
        { rec(m);
         td_vectj[i]=1};
Case 2. If (m.pr_csni<csn[i]; rec (m)); Pi has taken some permanent checkpoint
 // after sending m

Case 3. If(( m.pr_csni>csn[i])  AND (pr_csnj>csn[j]))
 {rec (m); td_vectj[i]=1}//Pi & Pj, both, have taken their tentative
 //checkpoints
Case 4. If(( m.pr_csni>csn[i])  AND (pr_csnj=csn[j]))
{Pj log m-1 } Pi has taken its tentative checkpoint
 // before sending m while Pj has not.
```

2.4.4 Algorithm executed at any MSS (say MSSs)

```
1. Wait for Response
2. Upon receiving message c_req (Pin, MSSin, p_csni, m_vect) from MSSin
 2.1 For any Pi such that mss_plists[i]=1∧ m_vect[i]=1; send c_req toPi
 2.2 ++pr_csni; mss_chk_request[i]=1, chkpt_sti=1
 2.3 Compute m_vect1, m_vect2, m_vect3 //Refer Section 4.1
 2.4 If ∃ i such that m_vect1[i]=1;
   send c_req to Pi.   //m_vect1 contains the new processes found for the //minimum set
3. On receiving c_req from some other MSS say MSSp
∀i such that(( mssp. m_vect1[i] =  1)
∧ (mss_p_mss[i]= 1) ∧ (mss_chk_req=1))
{ send c_req to Pi; compute m_vect1, m_vect2, m_vect3}
If ∃ j such that m_vect1[j]=1;
send c_req to Pj;
∀i, m_vect1[i]=0;
4. On receiving response to checkpointing from Pj
    4.1 If (Pj has taken the mutable checkpoint successfully the mss_chk_taken[j]=1 else mss_set fail_bit.)
    4.2 If (mss_fail_bit) ∨ (∀j mss_chk_taken[j]=mss_chk_request[j];
Send response (Pin, MSSin,msss, mss_chk_taken, mss_fail_bit, m_vect2) to MSSin;
5. On receiving tantative checkpoint request from MSSin tent_req(Pin, _MSSin ,pr_csn, m_vect[])
a) Send tentative checkpoint request to all process in its all which are in m_vect[] and store such processor in MSS_Chk_request2[]
For(k=0;k<n;n++)
{
If (m_vent[u]==s and MSS_plist[h]==1
Then(send tentative checkpoint report 2[h]=1);
b) on receiving positives response to checkpoint from Pj
if (Pj has taken its tentative checkpoint successfully, MSS_chk_taken2{j]=1
else
set MSS_fail_bit
if((mss_fail_bit)OR(for all j mss_chk_taken2[j]=mss_chk_request2[j])
send response (pin, mssin,msss,mss_chk_taken2, mss_fail_bit) to mssin)
6. On receiving commit().
Convert the tentative checkpoints in to permanent ones and discard old permanent checkpoints.if any.
∀j such that m_vect[j]=1, csn[j]++;
Initialize relevant data structures.
7. On receiving abort().
Discard the tentative checkpoints, if any.
Update relevant variables.
```

2.4. 5 Algorithm executed at any process Pi

On receiving tentative/mutable checkpoint request;
Take tentative/mutable checkpoint and inform local MSS.

2.8 Handling Node Mobility and Disconnections

MHs are typically powered by battery. From time to time, MHs may turn to doze mode or get disconnected with the network to save battery power. The duration of disconnection can be arbitrarily long and if a disconnected MH is involved in the checkpointing operation, then the checkpointing operation may have to wait for a long time or the operation must be aborted. To seamlessly execute the coordinated checkpoint collection algorithm, these situations needs to be taken care efficiently.
We, hereby, propose the following strategy to handle the above undesirable situations in the mobile systems during checkpointing operation. When a MH is disconnected from the cell of its MSS then it takes a local checkpoint and saves it with the MSS. This local checkpoint is saved in the same manner as it saves in normal situations on

receiving the checkpointing request from the initiator process. All the concerned data structures related with the MH are also saved on the MSS. During the disconnection, if a checkpoint request arrives for the MH then the MSS will execute the algorithm for the disconnected MH and will convert its local checkpoint (which was saved on MSS by MH before disconnection) in to tentative checkpoint and on getting the commit request will convert this tentative checkpoint into permanent checkpoint. If the messages are received for the disconnected MHs then the MSS will buffer all the messages in FIFO queue.

On reconnection, if the MH is not connected with the original MSS, then it first contact the original MSS and downloads all the data structures which were sent by this MH before disconnection. It also downloads all the messages which were buffered by the original MSS during the period of disconnection. The MH then processes these buffered messages in the same order in which they were received by the original MSS.

2.9 Handling Failures during checkpointing

An MH may fail during checkpointing process. If an MH fails after taking its tentative checkpoint or if it is not a member of minimum set, then the checkpointing procedure can be completed uninterruptedly. If a process fails during checkpointing, then our straight forward approach is to discard the whole checkpointing operation . The failed process will not be able to respond to the initiator's request and the initiator will detect the failure by timeout and will discard the complete checkpointing operation. If the initiator fails after sending commit, the checkpointing process can be considered complete. If the initiator fails during checkpointing, then some processes, waiting for commit will time out and will issue abort on his own.

Kim and Park [6] proposed that a process commits its tentative checkpoints if none of the processes, on which it transitively depends, fails; and the consistent recovery line is advanced for those processes that committed their checkpoints. The initiator and other processes, which transitively depend on the failed process, have to abort their tentative checkpoints. Thus, in case of a node failure during checkpointing, total abort of the checkpointing is avoided.

3. PERFORMANCE EVALUATION

3.1General Comparison with existing minimum process algorithms:

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.

3.2 Comparison with other Algorithms:

We use following notations to compare our algorithm with other algorithms:

Nmss: number of MSSs.

Nmh: number of MHs.

Cpp: cost of sending a message from one process to another

Cst: cost of sending a message between any two MSSs.

Cwl: cost of sending a message from an MH to its local MSS (or vice versa).
Cbst: cost of broadcasting a message over static network.
C_{search}: *cost incurred to locate an MH and forward a message to its current local MSS, from a source MSS.*
T_{st}: average message delay in static network.
T_{wl}: average message delay in the wireless network.
T_{ch}: average delay to save a checkpoint on the stable storage. It also includes the time to transfer the checkpoint from an MH to its local MSS.
N: total number of processes
N_{min}: number of minimum processes required to take checkpoints.
N_{mut}: number of useless mutable checkpoints [5].
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 [5].
N_{dep}: average number of processes on which a process depends.
h_1: height of the checkpointing tree in Koo-Toueg algorithm [11].
h_2: height of the checkpointing tree in the proposed algorithm.
In the algorithm [11], [5] and in the proposed one, the checkpoint initiator process, say P_{in} sends the checkpoint request to any process P_i if P_{in} is causally dependent upon P_i. Similarly, P_i sends the checkpoint request to any process P_j if P_i is causally dependent upon P_j. In this way, a checkpointing tree is formed.

3.2.1 Performance of our algorithm

The Synchronization message overhead:

Initiator process sends request and response to its local MSS: $2C_{wl}$
Initiator MSS broadcasts mutable |tentative |commit request over static network: $3C_{bst}$
Every process in the minimum set receives mutable checkpoint request, tentative checkpoint request from the local MSS and it also sends responses to these requests to local MSS: $4*N_{min}*C_{wl}$
Every MSS sends mutable checkpoint and tentative checkpoint response to initiator MSS : $2*N_{mss}*C_{st}$
Total Average overhead = 2Cwl+3 bst +$4*N_{min}*C_{wl}$ + $2*N_{mss}*C_{st}$
In our algorithm, anti-messages of very few processes are blocked during checkpointing at receiver end. The loss of checkpointing effect is reduced in case of abort in the first phase.

***Number of processes taking checkpoints*:** In our algorithm, only minimum numbers of processes are required to take their checkpoints.
The blocking time of the Koo-Toueg [11] protocol is highest, followed by Cao-Singhal [4] algorithm. In the algorithms proposed in [5], [19], [20], no blocking of processes takes place, but some useless checkpoints are taken, which are discarded on commit. In Elnozahy et al [8] algorithm, all processes take checkpoints. In the protocols [4], [11], and in the proposed one, only minimum numbers of processes record their checkpoints. In algorithm [5], concurrent executions of the algorithm are allowed, but it may lead to inconsistencies in doing so [20]. We avoid the concurrent executions of the proposed algorithm. We store anti-messages of very few messages at receiver end only during the checkpointing period.
The message overhead in our algorithm is larger than two-phase algorithms as our algorithm is a three phase algorithm. In the first phase imitator

	Cao-Singhal [4]	Cao-Singhal [5]	Koo-Toeg [11]	Elnozahy et al [8]	Propos-ed Algorit-hm
Avg. block Time	2Tst	0	h1*Tch	0	0
Avg No. of checkpoints	Nmin	Nmin+ Nmut	Nmin	N	Nmin
Avg Messa-ge Over-head	3Cbst+ 2Cwireless +2Nmss*Cst +3Nmh * Cwl	2Nmin *Cpp +Cbst+ Nucr* Cpp	3*Nmin *Cpp * Ndep	2Cbst + N *Cpp	2Cwl +3 bst + 4Nmin *Cwe + 2Nmss *Cst

Table 1 A Comparison of System Performance

broadcast tentative minimum set over the static network and all processes in the minimum set send their response to the initiator. In this way extra message overhead as compared to other two phase algorithms is $C_{bst} + 2*N_{min} * C_{wl}$. By increasing this message overhead, we try to reduce the loss of checkpointing overhead due to abort of checkkpointing protocol in the first phase. Because, in case of an abort of checkpointing algorithm in the first phase all processes need to abort their tentative checkpoints in two phase algorithms. But, in our algorithm processes need to abort their mutable checkpoints only. The effort of taking a mutable checkpoint is negligibly small as compared to tentative checkpoint[5].
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.

4.9 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].

REFRENCES:

[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 *International Journal of Computer Applications (0975 – 8887)Volume 3 – No.1, June 2010* 27 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 Nonintrusive 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 Minimumprocess Coordinated Checkpointing Protocol for Mobile Computing Systems", *International Journal of Foundations of Computer science*, Vol 19, No. 4, pp 1015-1038 (2008)