Dealing with Frequent Aborts in Minimum-process Coordinated Checkpointing Algorithm for Mobile Distributed Systems

Parveen Kumar
MIET
Department of CSE
Meerut (INDIA)- 250005

Preeti Gupta
Singhania University
Pacheri Bari (Jhunjhunu)
Rajasthan (India)

Anil Kumar Solanki
MIET
Department of CSE
Meerut (INDIA)- 250005

ABSTRACT

While dealing with mobile distributed systems, we come across some 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. In this paper, we design a minimum process algorithm for Mobile Distributed systems, where no useless checkpoints are taken and an effort has been made to optimize the blocking of processes. In order to keep the blocking time minimum, we collect the dependency vectors and compute the exact minimum set in the beginning of the algorithm. 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. In order to take its tentative checkpoint, an MH (Mobile Host) needs to transfer large checkpoint data to its local MSS over wireless channels. The checkpointing effort may be exceedingly high due to frequent aborts especially in mobile systems. We try to minimize the loss of checkpointing effort when any process fails to take its checkpoint in coordination with others

Key words: Fault tolerance**,** consistent global state, coordinated checkpointing and mobile systems.

1.BACKGROUND

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. 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. A distributed system consists of a finite set of processes and a finite set of channels.

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.

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. Checkpointing is the process of saving the status information. 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. The checkpoint-restart is one of the well-known methods to realize reliable distributed systems. Each process takes a checkpoint where the local state information is stored in the stable storage. Rolling back a process and again resuming its execution from a prior state involves overhead and delays the overall completion of the process, it is needed to make a process rollback to a most recent possible state. So it is at the desire of the user for taking many checkpoints over the whole life of the execution of the process [6].

In a distributed system, since the processes in the system do not share memory, a global state of the system is defined as a set of local states, one from each process. The state of channels corresponding to a global state is the set of messages sent but not yet received. A global state is said to be “consistent” if it contains no orphan message; i.e., a message whose receive event is recorded, but its send event is lost. To recover from a failure, the system restarts its execution from a previous consistent global state saved on the stable storage during fault-free execution. This saves all the computation done up to the last checkpointed state and only the computation done thereafter needs to be redone. In distributed systems, checkpointing can be independent, coordinated [6, 11, 13] or quasi-synchronous [2]. Message Logging is also used for fault tolerance in distributed systems [22].

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 [6, 11, 23]. In the first phase, processes take tentative checkpoints and in the second phase, these are made permanent. The main advantage is that only one permanent checkpoint and at most one tentative checkpoint is required to be stored. In the case of a fault, processes rollback to last checkpointed state.

The coordinated checkpointing protocols can be classified into two types: blocking and non-blocking. In blocking algorithms, some blocking of processes takes place during checkpointing [4, 11, 24, 25] In non-blocking algorithms, no blocking of processes is required for checkpointing [5, 12, 15, 21]. 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 [11].

In minimum-process coordinated checkpointing algorithms, a process P_i takes its checkpoint only if it a member of the minimum set (a subset of interacting process). A process P_i is in the minimum set only if the checkpoint initiator process is transitively dependent upon it. P_j is directly dependent upon P_k only if there exists m such that P_j receives m from P_k in the current checkpointing interval [CI] and P_k has not taken its permanent checkpoint after sending m. 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.

In minimum-process checkpointing protocols, some useless checkpoints are taken or blocking of processes takes place. In this paper, we propose a minimum-process coordinated checkpointing algorithm for non-deterministic mobile distributed systems, where no useless checkpoints are taken. An effort has been made to minimize the blocking of processes and the loss of checkpointing effort when any process fails to take its checkpoint in coordination with others.

2. INTRODUCTION

The proposed scheme is based on keeping track of direct dependencies of processes. Similar to [4], initiator process collects the direct dependency vectors of all processes, computes minimum set, and sends the checkpoint request along with the minimum set to all processes. In this way, blocking time has been significantly reduced as compared to [11].

During the period, when a process sends its dependency set to the initiator and receives the minimum set, may receive some messages, which may add new members to the already computed minimum set [25]. In order to keep the computed minimum set intact, We have classified the messages, received during the blocking period, into two types: (i) messages that alter the dependency set of the receiver process (ii) messages that do not alter the dependency set of the receiver process. The messages in point (i) need to be delayed at the receiver side [25]. The messages in point (ii) can be processed normally. All processes can perform their normal computations and send messages during their blocking period. When a process buffers a message of former type, it does not process any message till it receives the minimum set so as to keep the proper sequence of messages received. When a process gets the minimum set, it takes the checkpoint, if it is in the minimum set. After this, it receives the buffered messages, if any. The proposed minimum-process blocking algorithm forces zero useless checkpoints at the cost of very small blocking.

In minimum-process synchronous checkpointing, the initiator process asks all communicating processes to take tentative checkpoints. In this scheme, 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. Due to frequent aborts, total checkpointing effort may be exceedingly high, which may be undesirable in mobile systems due to scarce resources. Frequent aborts may happen in mobile systems due to exhausted battery, abrupt disconnection, or bad wireless connectivity. Therefore, we propose that in the first phase, all concerned MHs will take mutable checkpoint only. Mutable 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 mutable checkpoints only. The effort of taking a mutable checkpoint is negligible as compared to the tentative one. When the initiator 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. In this way, by increasing small synchronization message overhead, we try to reduce the total checkpointing effort.

Our system model is similar to [5, 24, 25]. 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. 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.

3. THE PROPOSED CHECKPOINTING ALGORITHM

3.1 The Minimum-process Coordinated Checkpointing Scheme

The initiator MSS sends a request to all MSSs to send the *dd_set* vectors of the processes in their cells. All *dd_set* vectors are at MSSs and thus no initial checkpointing messages or responses travels wireless channels. On receiving the *dd_set* [] request, an MSS records the identity of the initiator process (say mss_id_a) and initiator MSS, sends back the *dd_set* [] of the processes in its cell, and sets *g_chkpt*. If the initiator MSS receives a request for *dd_set* [] from some other MSS (say mss_id_b) and mss_id_a is lower than mss_id_b,the, current initiation with mss_id_a is discarded and the new one having mss_id_b is continued. Similarly, if an MSS receives *dd_set* requests from two MSSs, then it discards the request of the initiator MSS with lower mss_id. Otherwise, on receiving *dd_set* vectors of all processes, the initiator MSS computes *min_vect* [], sends mutable checkpoint request along with the *min_vect* [] to all MSSs. When a process sends its *dd_set* [] to the initiator MSS, it comes into its blocking state. A process comes out of the blocking state only after taking its mutable checkpoint if it is a member of the minimum set; otherwise, it comes out of blocking state after getting the mutable checkpoint request.

On receiving the mutable checkpoint request along with the *min_vect* [], an MSS, say MSS_j, takes the following actions. It sends the mutable checkpoint request to P_i only if P_i belongs to the *min_vect* [] and P_i is running in its cell. On receiving the checkpoint request, P_i takes its mutable checkpoint and informs MSS_j. On receiving positive response from P_i, MSS_j updates $p\text{-}csn_i$, resets $blocking_i$, and sends the buffered messages to P_i, if any. Alternatively, If P_i is not in the *min_vect* [] and P_i is in the cell of MSS_j, MSS_j resets $blocking_i$ and sends the buffered message to P_i, if any. For a disconnected MH, that is a member of *min_vect* [], the MSS that has its disconnected checkpoint, converts its disconnected checkpoint into the required one.
During blocking period, P_i processes *m*, received from P_j, if following conditions are met: (i) (!$bufer_i$) i.e. P_i has not buffered any message (ii) ($m.psn$ <=$csn[j]$) i.e. P_j has not taken its checkpoint before sending *m* (iii) (dd_set_i[j]=1) P_i is already dependent upon P_j in the current CI or P_j has taken some permanent checkpoint after sending *m*.
Otherwise, the local MSS of P_i buffers *m* for the blocking period of P_i and sets $buffer_i$.

When an MSS learns that all of its processes in minimum set have taken their mutable checkpoints or at least one of its process has failed to checkpoint, it sends the response message to the initiator MSS. In this case, if some process fails to take mutable checkpoint in the first phase, then MHs need to abort their mutable checkpoints only. The effort of taking a mutable checkpoint is negligible as compared to the tentative one. When the initiator 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.

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

3.2 An Example

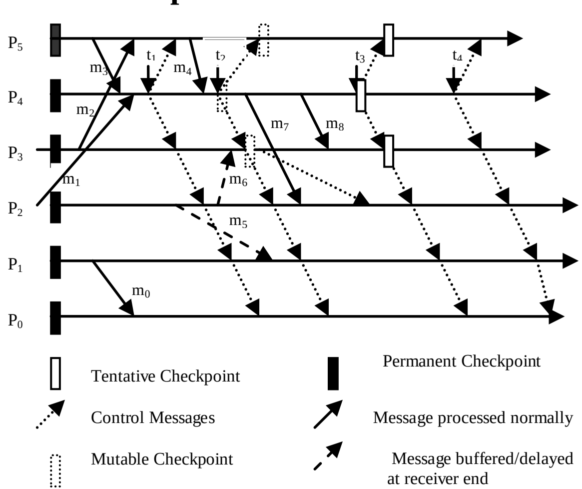

Figure 1 An Example of the proposed Protocol

We explain the proposed minimum-process checkpointing algorithm with the help of an example. In Figure 1, at time t_1, P_4 initiates checkpointing process and sends request to all processes for their dependency vectors. At time t_2, P_4 receives the dependency vectors from all processes (not shown in the Figure 1) and computes the minimum set (*min_vect[]*) which is {P_3, P_4, P_5}. P_4 sends *min_vect[]*to all processes and takes its own mutable checkpoint. A process takes its mutable checkpoint if it is a member of *min_vect[]*. When P_3 and P_5 get the *min_vect*[], they find themselves in the *min_vect*[]; therefore, they take their mutable checkpoints. When P_0, P_1 and P_2 get the *min_vect* [], they find that they do not belong to *min_vect* [], therefore, they do not take their mutable checkpoints.

A process comes into the blocking state immediately after sending the *dd_set[]*. A process comes out of the blocking state only after taking its mutable checkpoint if it is a member of the minimum set; otherwise, it comes out of blocking state after getting the mutable checkpoint request. P_4 receives m_4 during its blocking period. As dd_set_4[5]=1 due to m_3, and receive of m_4 will not alter dd_set_4[]; therefore P_4 processes m_4. P_1 receives m_5 from P_2 during its blocking period; dd_set_1[2]=0 and the receive of m_5 can alter dd_set_1[]; therefore, P_1 buffers m_5. Similarly, P_3 buffers m_6. P_3 processes m_6 only after taking its mutable checkpoint. P_1 process m_5 after getting the *min_vect* [].P_2 processes m_7 because at this movement it not in the blocking state. Similarly, P_3 processes m_8. At time t_3, P_4 receives responses to mutable check point requests from all relevant processes (not shown in the Figure 1) and issues tentative checkpoint request to all processes. A process in the minimum set converts its mutable checkpoint into tentative one. Finally, at time t_4, P_4 receives responses to tentative checkpoint requests from all relevant processes (not shown in the Figure 1) and issues the commit request.

3.3 Handling Node Mobility and Disconnections

An MH may be disconnected from the network for an arbitrary period of time. The Checkpointing algorithm may generate a request for such MH to take a checkpoint. Delaying a response may significantly increase the completion time of the checkpointing algorithm. We propose the following solution to deal with disconnections that may lead to infinite wait state.

When an MH, say MH_i, disconnects from an MSS, say MSS_k, MH_i takes its own checkpoint, say $disconnect\_ckpt_i$, and transfers it to MSS_k. MSS_k stores all the relevant data structures and $disconnect\_ckpt_i$ of MH_i on stable storage. During disconnection period, MSS_k acts on behalf of MH_i as follows. In minimum-process checkpointing, if MH_i is in the *minset[]*, $disconnect\_ckpt_i$ is considered as MH_i's checkpoint for the current initiation. In all-process checkpointing, if MH_i's $disconnect\_ckpt_i$ is already converted into permanent one, then the committed checkpoint is considered as the checkpoint for the current initiation; otherwise, $disconnect\_ckpt_i$ is considered. On global checkpoint commit, MSS_k also updates MH_i's data structures, e.g., *ddv*[], *cci* etc. On the receipt of messages for MH_i, MSS_k does not update MH_i's *ddv*[] but maintains two message queues, say *old_m_q* and *new_m_q*, to store the messages as described below.

On the receipt of a message *m* for MH_i at MSS_k from any other process:

if((m.*cci*= = cci_i ∨ (*m.cci*= =nci_i) ∨ (*matd*[j, *m.cci*]= =1))
add (*m*, *new_m_q*); // keep the message in new_m_q
else
add(*m*, old_m_q);

On all-process checkpoint commit:
Merge *new_m_q* to *old_m_q*;
Free(*new_m_q*);

When MH_i, enters in the cell of MSS_j, it is connected to the MSS_j if g_chkpt_j is reset. Otherwise, it waits for g_chkpt_j to be reset. Before connection, MSS_j collects MH_i's *ddv*[], *cci*, *new_m_q*, *old_m_q* from MSS_k; and MSS_k discards MH_i's support information and $disconnect\_ckpt_i$. MSS_j sends the messages in *old_m_q* to MH_i without updating the *ddv*[], but messages in *new_m_q*, update *ddv*[] of MH_i.

Handling Failures during Checkpointing

Since MHs are prone to failure, an MH may fail during checkpointing process. Sudden or abrupt disconnection of an MH is also termed as a fault. Suppose, P_i is waiting for a message from P_j and P_j has failed, then P_i times out and detects the failure of P_j. If the failed process is not required to checkpoint in the current initiation or the failed process has already taken its tentative checkpoint, the checkpointing process can be completed uninterruptedly. If the failed process is not the initiator, one way to deal with the failure is to discard the whole checkpointing process similar to the approach in [11, 21]. The failed process will not be able to respond to the initiator's requests and initiator will detect the failure by timeout and will abort the current checkpointing process. If the initiator fails after sending *commit* or *abort* message, it has nothing to do for the current initiation. Suppose, the initiator fails before sending *commit* or *abort* message. Some process, waiting for the checkpoint/commit request, will timeout and will detect the failure of the initiator. It will send *abort* request to all processes discarding the current checkpointing process.

The above approach seems to be inefficient, because, the whole checkpointing process is discarded even when only one participating process fails. Kim and Park [13] 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.

Multiple Concurrent Initiations

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

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

(ii) In case of concurrent initiations, multiple triggers need to be piggybacked on normal messages [26]. Trigger contains the initiator process identification and its csn. This leads to considerable increase in piggybacked information.

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

Correctness Proof

The correctness proof for the proposed minimum-process checkpointing algorithm is as under:
Let $GC_i=\{C_{1,x}, C_{2,y},............,C_{n,z}\}$ be some consistent global state created by our algorithm, where $C_{i,x}$ is the x^{th} checkpoint of P_i.

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

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

Case I: *P_i belongs to minimum set and P_j does not:*

As P_i is in minimum set, $C_{i,x}$ is the checkpoint taken by P_i during the current initiation and $C_{j,y}$ is the checkpoint taken by P_j during some previous initiation i.e. $C_{j,y} \rightarrow C_{i,x}$. Therefore rec(*m*) $\rightarrow C_{j,y}$ and $C_{i,x} \rightarrow$send(*m*) implies rec(*m*) $\rightarrow C_{j,y} \rightarrow C_{i,x} \rightarrow$send(*m*) implies rec(*m*) $\rightarrow$send(*m*) which is not possible. '→' is the Lamport's happened before relation [17].

Case II: *Both P_i and P_j are in minimum set:*

Both $C_{i,x}$ and $C_{j,y}$ are the checkpoints taken during current initiation. There are following possibilities:
(a) P_i sends m after taking the tentative/mutable checkpoint and P_j receives m before receiving request for dependency: Any process can take the checkpoint only after initiator receives the dependencies from all processes. Therefore a message sent from a process after taking the checkpoint can not be received by other process before getting the dependency request.

(b)P_i sends m after taking the mutable checkpoint and P_j receives m after getting the dependency request but before taking the checkpoint:

In this case, following condition will be true at the time of receiving *m*: (*blocking*$_j$) && (*m.p_csn* >csn[j])). Therefore, *m* will be buffered at P_j, and it will be processed only after P_j takes the mutable checkpoint.

(c) P_i sends m after commit and P_j receives m before taking tentative checkpoint:

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

Case III: *P_i is not in minimum set but P_j is in minimum set:*

Checkpoint $C_{j,y}$ belongs to the current initiation and $C_{i,x}$ is from some previous initiation. The message *m* can be received by P_j:

(i) before receiving request for dependency

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

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

a. *dd_set*$_j$[i]=1. In this case P_i will also be included in the minimum set.

b. (*m.p_csn* >csn[i]). This is possible only if P_i has taken some permanent checkpoint after sending *m*. In that case, *m* is not an orphan message.

c.

Case IV: *Both P_i and P_j are not in minimum set:*

Neither P_i nor P_j will take a new checkpoint, therefore, no such *m* is possible unless and until it already exists.

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

4. CONCLUSION

We have proposed a minimum process coordinated checkpointing algorithm for mobile distributed system, where no useless checkpoints are taken and an effort is made to minimize the blocking of processes. We are able to reduce the blocking time to bare minimum by computing the exact minimum set in the beginning. Furthermore, the blocking of processes is reduced by allowing the processes to perform their normal computations and send messages during their blocking period. The number of processes that take checkpoints is minimized to avoid awakening of MHs in doze mode of operation and thrashing of MHs with checkpointing activity. It also saves limited battery life of MHs and low bandwidth of wireless channels. We try to reduce the loss of checkpointing effort when any process fails to take its checkpoint in coordination with others.

5. REFERENCES

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

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

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

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

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

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

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

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

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

[10] H. Higaki and M. Takizawa, Checkpoint-recovery Protocol for Reliable Mobile Systems, *Transactions of Information processing Japan*, 40(1), 1999, 236-244.

[11] R. Koo and S. Toueg, Checkpointing and Roll-Back Recovery for Distributed Systems, *IEEE Transactions on Software Engineering*, 13(1), 1987, 23-31.

[12] P. Kumar, L. Kumar, R. K. Chauhan and V. K. Gupta, *A Non-Intrusive Minimum Process*

Synchronous Checkpointing Protocol for Mobile Distributed Systems, In Proceedings of IEEE ICPWC-2005, 2005.

[13] J.L. Kim and T. Park, An efficient Protocol for checkpointing Recovery in Distributed Systems, *IEEE Transactions on Parallel and Distributed Systems*, 1993, 955-960.

[14] L. Kumar, M. Misra, R.C. Joshi, Checkpointing in Distributed Computing Systems, In Concurrency in Dependable Computing, 2002, 273-92.

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

[16] L. Kumar and P.Kumar, A Synchronous Checkpointing Protocol for Mobile Distributed Systems: Probabilistic Approach, *International Journal of Information and Computer Security*, 1(3), 2007, 298-314.

[17] L. Lamport, Time, clocks and ordering of events in a distributed system, *Communications of the ACM*, 21(7), 1978, 558-565.

[18] N. Neves and W.K. Fuchs, Adaptive Recovery for Mobile Environments, *Communications of the ACM*, 40(1), 1997, 68-74.

[19] W. Ni, S. Vrbsky and S. Ray, Pitfalls in Distributed Nonblocking Checkpointing, *Journal of Interconnection Networks*, 1(5), 2004, 47-78.

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

[21] R. Prakash and M. Singhal, Low-Cost Checkpointing and Failure Recovery in Mobile Computing Systems, *IEEE Transaction On Parallel and Distributed Systems*, 7(10), 1996, 1035-1048.

[22] K.F. Ssu, B. Yao, W.K. Fuchs and N.F. Neves, Adaptive Checkpointing with Storage Management for Mobile Environments, *IEEE Transactions on Reliability*, 48(4), 1999, 315-324.

[23] L.M. Silva and J.G. Silva, *Global checkpointing for distributed programs*, In Proceedings of the 11th symposium on Reliable Distributed Systems, 1992, 155-62.

[24] 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).

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

[26] W. Ni, S. Vrbsky and S. Ray, Pitfalls in Distributed Nonblocking Checkpointing, *Journal of Interconnection Networks*, 1(5), 2004, 47-7