A Low-Overhead Minimum Process Coordinated Checkpointing Algorithm for Mobile Distributed System

Parveen Kumar[1], Poonam Gahlan[2]
[1]Department of Computer Science & Engineering
Meerut Institute of Engineering & Technology, Meerut, India, -250005
[2]Department of Computer Sc & Engg, Singhania University, Pacheri Bari (Rajasthan) India

ABSTRACT
A distributed system is a collection of independent entities that cooperate to solve a problem that cannot be individually solved. A mobile computing system is a distributed system where some of processes are running on mobile hosts (MHs), whose location in the network changes with time. 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. 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 synchronization message overhead. We try to reduce the loss of checkpointing effort when any process fails to take its checkpoint in coordination with others.

KEYWORDS: Checkpointing algorithms; parallel & distributed computing; rollback recovery; fault-tolerant system; mobile computing.

1. INTRODUCTION
A distributed system is a collection of independent entities that cooperate to solve a problem that cannot be individually solved. With the widespread proliferation of the Internet and the emerging global village, the notion of distributed computing systems as a useful and widely deployed tool is becoming a reality [24]. A distributed system can be characterized as a collection of mostly autonomous processors communicating over a communication network and having the following features [25]:

• **No common physical clock** This is an important assumption because it introduces the element of "distribution" in the system and gives rise to the inherent asynchrony amongst the processors.

. No shared memory This is a key feature that requires message-passing for communication. It may be noted that a distributed system may still provide the abstraction of a common address space via the distributed shared memory abstraction.

• **Geographical separation** It is not necessary for the processors to be on a wide-area network (WAN). Recently, the network/cluster of workstations (NOW/COW) configuration connecting processors on a LAN is also being increasingly regarded as a small distributed system. This NOW configuration is becoming popular because of the low-cost high-speed off-the-shelf processors now available. The Google search engine is based on the NOW architecture.

• **Autonomy and heterogeneity** The processors are "loosely coupled in that they have different speeds and each can be running a different operating system. They are usually not part of a dedicated system, but cooperate with one another by offering services or solving a problem [25].

Local checkpoint is the saved state of a process at a processor at a given instance. Global checkpoint is a collection of local checkpoints, one from each process. 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. In distributed systems, checkpointing can be independent, coordinated or quasi-synchronous. Message Logging is also used for fault tolerance in distributed systems [14]. Most of the existing coordinated checkpointing algorithms [9, 19] rely on the two-phase protocol and save two kinds of checkpoints on the stable storage: tentative and permanent. In the first phase, the initiator process takes a tentative checkpoint and requests all or selective processes to take their tentative checkpoints. If all processes are asked to take their checkpoints, it is called all-process coordinated checkpointing [5, 7, 19]. Alternatively, if selective communicating processes are required to take checkpoints, it is called minimum-process checkpointing. Each process informs the initiator whether it succeeded in taking a tentative checkpoint. After the initiator has received positive acknowledgments from all relevant processes, the algorithm enters the second phase. Alternatively, if a process fails to take its tentative checkpoint in the first phase, the initiator process requests all processes to abort their tentative checkpoint.

If the initiator learns that all concerned processes have successfully taken their tentative checkpoints, the algorithm enters in the second phase and the initiator asks the relevant processes to make their tentative checkpoints permanent. In

order to record a consistent global checkpoint, when a process takes a checkpoint, it asks (by sending checkpoint requests to) all relevant processes to take checkpoints. Therefore, coordinated checkpointing suffers from high overhead associated with the checkpointing process [20], [21], [22], [23]. Much of the previous work [2, 3, 4, 20, 21, 22, 23] in coordinated checkpointing has focused on minimizing the number of synchronization messages and the number of checkpoints during the checkpointing process. However, some algorithms (called blocking algorithm) force all relevant processes in the system to block their computations during the checkpointing process [3, 9, 21, 22, 23]. Checkpointing includes the time to trace the dependency tree and to save the states of processes on the stable storage, which may be long. Moreover, in mobile computing systems, due to the mobility of MHs, a message may be routed several times before reaching its destination. Therefore, blocking algorithms may dramatically reduce the performance of these systems [7]. Recently, non-blocking algorithms [7, 19] have received considerable attention. In these algorithms, processes need not block during the checkpointing by using a checkpointing sequence number to identify orphan messages. Moreover, these algorithms [4, 10] require all processes in the system to take checkpoints during checkpointing, even though many of them may not be necessary.

A mobile computing system is a distributed system where some of processes are running on mobile hosts (MHs), whose location in the network changes with time. To communicate with MHs, mobile support stations (MSSs) act as access points for the MHs by wireless networks. Features that make traditional checkpointing algorithms for distributed systems unsuitable for mobile computing systems are: locating processes that have to take their checkpoints, energy consumption constraints, lack of stable storage in MHs, and low bandwidth for communication with MHs [1]. Minimum-process coordinated checkpointing is an attractive approach for transparently adding fault tolerance to distributed applications, since it avoids domino effect, minimizes the stable storage requirement and also forces only interacting processes to checkpoint.

Prakash-Singhal algorithm [13] forces only a minimum number of processes to take checkpoints and does not block the underlying computation during checkpointing. However, it was proved that their algorithm may result in an inconsistency [3]. Cao and Singhal [4] achieved non-intrusiveness in the minimum-process algorithm by introducing the concept of mutable checkpoints. The number of useless checkpoints in [4] may be exceedingly high in some situations [16]. Kumar et. al [16] and Kumar et. al [11] 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. Some minimum-process blocking algorithms are also proposed in literature [3, 9, 21, 23]. In this paper, we propose an efficient checkpointing algorithm for mobile computing systems that forces only a minimum number of processes to take checkpoints. An effort has been made to minimize the blocking of processes and synchronization message overhead. We capture the partial transitive dependencies during the normal execution by piggybacking dependency vectors onto computation messages. The Z-dependencies are well taken care of in this protocol. In order to reduce the message overhead, we also avoid collecting dependency vectors of all processes to find the minimum set as in [3], [11], [21]. We also try to minimize the loss of checkpointing effort when any process fails to take its checkpoint.

2. PROPOSED CHECKPOINTING ALGORITHM

Our system model is similar to [4, 21]. We propose to handle node mobility and failures during checkpointing as proposed in [21].

2.1 BASIC IDEA

All Communications to and from MH pass through its local MSS. The MSS maintains the dependency information of the MHs which are in its cell. The dependency information is kept in Boolean vector Ri for process Pi. The vector has n bits for n processes. When Ri[j] is set to 1, it represents Pi depends upon Pj. For every Pi, Ri is initialized to 0 except Ri[i], which is initialized to l. When a process Pi running on an MH, say MHp, receives a message from a process Pj, MHp's local MSS should set Ri[j] to 1.If PJ has taken its permanent checkpoint after sending Ri[j] is not updated. Suppose there are processes Pi and Pj running on MHs, MHi and MHj with dependency vectors Ri and Rj. The dependency vectors of MHs, MHi and MHj are maintained by their local MSSs, MSSi and MSSj. Process Pi running on MHi sends message m to process Pj running on MHj. The message is first sent to MSSi (local MSS of MHi). MSSi maintains the dependency vector Ri of MHi. MSSi appends Ri with message m and sends it to MSSj (local MSS of MHj). MSSj maintains the dependency vector Rj of MHj. MSSj replaces Rj with bitwise logical OR of dependency vectors Ri and Rj and sends m to Pj.

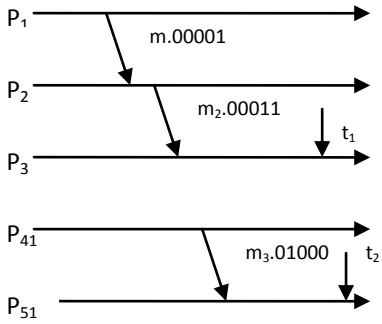

Figure1. Maintenance of Dependency Vectors

In Figure 1, there are five processes P_1, P_2, P_3, P_4, P_5 with dependency vectors R_1, R_2, R_3, R_4, R_5 initialized to 00001, 00010, 00100, 01000, and 10000 respectively. Initially,

every process depends upon itself. Now process P_1 sends m to P_2. P_1 appends R_1 with m. P_2 replaces R_2 with the bitwise logical OR of R_1(00001) and R_2(00010), which comes out to be (00011). Now P_2 sends m2 to P_3 and appends R_2 (00011) with m_2. Before receiving m_2, the value of R_3 at P_3 was 00100. After receiving m_2, P_3 replaces R_3 with the bitwise logical OR of R_2 (00011) and R_3 (00100) and R_3 becomes (00111). Now P_4 sends m_3 along with R_4 (01000) to P_5. After receiving m_3, R_5 becomes (11000).In this case, if P_3 starts checkpointing at t_1, it will compute the tentative minimum set equivalent to R_3(00111), which comes out to be $\{P_1, P_2, P_3\}$. If a single process fails to take its checkpoint; all the checkpointing effort goes waste, because, each process has to abort its tentative checkpoint. Furthermore, in order to take the tentative checkpoint, an MH needs to transfer large checkpoint data to its local MSS over wireless channels. Hence, the loss of checkpointing effort may be exceedingly high due to frequent aborts of checkpointing algorithms. In mobile distributed systems, there remain certain issues like: abrupt disconnection, exhausted battery power, or failure in wireless bandwidth. So there remains a good probability that some MH may fail to take its checkpoint in coordination with others. Therefore, we propose that in the first phase, all processes in the minimum set, take mutable checkpoint only. Mutable checkpoint is described in [4], it is stored on the memory of MH only. If some process fails to take its checkpoint in the first phase, then other MHs need to abort their mutable checkpoints only. The effort of taking a mutable checkpoint is negligible as compared to the tentative one [4]. In this second phase, a process converts its mutable checkpoint into tentative one. By using this scheme, we try to minimize the loss of checkpointing effort in case of abort of checkpointing algorithm in the first phase.

A non-blocking checkpointing algorithm does not require any process to suspend its underlying computation. When processes do not suspend their computation, it is possible for a process to receive a computation message from another process, which is already running in a new checkpointing interval. If this situation is not properly dealt with, it may result in an inconsistency. During the checkpointing procedure, a process P_i may receive m from P_j such that P_j has taken its 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 only those messages, which can become orphan, should be buffered at the sender's end. When a process takes its mutable checkpoint, it is not allowed to send any message till it receives the tentative checkpoint request. However, in this duration, the process is allowed to perform its normal computations and receive the messages. When a process receives the tentative checkpoint request, it is confirmed that every concerned process has taken its mutable checkpoint. Hence, a message generated for sending by a process after taking its mutable checkpoint can not become orphan.

2.2 THE PROPOSED ALGORITHM

First phase of the algorithm: When a process, say P_i, running on an MH, say MH_i, initiates a checkpointing, it sends a checkpoint initiation request to its local MSS, which will be the proxy MSS (if the initiator runs on an MSS, then the MSS is the proxy MSS). The proxy MSS maintains the dependency vector of P_i say R_i. On the basis of R_i, the set of dependent processes of P_i is formed, say S_{minset}. The proxy MSS broadcasts ckpt (S_{minset}) to all MSSs. When an MSS receive ckpt (S_{minset}) message, it checks, if any processes in S_{minset} are in its cell. If so, the MSS sends mutable checkpoint request message to them. Any process receiving a mutable checkpoint request takes a mutable checkpoint and sends a response to its local MSS. After an MSS received all response messages from the processes to which it sent mutable checkpoint request messages, it sends a response to the proxy MSS. 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[4]. 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.

Second Phase of the Algorithm: After the proxy MSS has received the response from every MSS, the algorithm enters the second phase. If the proxy MSS learns that all relevant processes have taken their mutable checkpoints successfully, it asks them to convert their mutable checkpoints into tentative ones and also sends the exact minimum set along with this request. Alternatively, if initiator MSS 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 MSS receives the tentative checkpoint request, it asks all the process in the minimum set, which are also running in itself, to convert their mutable checkpoints into tentative ones. When an MSS learns that all relevant process in its cell have taken their tentative checkpoints successfully, it sends response to proxy MSS.

Third Phase of the Algorithm:

Finally, when the proxy 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 its tentative checkpoint into permanent one and discards its earlier permanent checkpoint, if any.

Message handling during checkpointing:

When a process takes its mutable checkpoint, it does not send any massage till it receives the tentative checkpoint request. Suppose, P_i sends m to P_j after taking its mutable checkpoint and P_j has not taken its mutable checkpoint at the time of receiving m. In this case, if P_j takes its mutable checkpoint after processing m, then m will become orphan. Therefore, we do not allow Pi to send any massage unless and until every process in the minimum set have taken its mutable checkpoint in the first phase. P_i can send massages when it receives the tentative checkpoint request; because, at this moment every concerned process has taken its mutable checkpoint and m cannot become orphan. The massages to be sent are buffered at senders end. In this duration, a process is allowed to continue its normal computations and receive massages.

Suppose, P_j gets the mutable checkpoint request at MSS_p. Now, we find any process P_k such that P_k does not belong to S_{minset} and P_k belongs to $R_j[]$. In this case, P_k is also included in the minimum set; and P_j sends mutable checkpoint request to P_k. It should be noted that the S_{minset}, computed on the basis of dependency vector of initiator process is only a subset of the minimum set. Due to zigzag dependencies, initiator process may be transitively dependent upon some more process which is not included in the S_{minset}.

2.3 AN EXAMPLE

The proposed Algorithm can be better understood by the example shown in Figure 2. There are six processes (P_0 to P_5) denoted by straight lines. Each process is assumed to have initial permanent checkpoints with csn equal to "0". C_{ix} denotes the x^{th} checkpoints of P_i. Initial dependency vectors of P_0, P_1, P_2, P_3, P_4, P_5 are [000001], [000010] [000100], [001000], [010000], and [100000], respectively. The dependency vectors are maintained as explained in Section 2.1. P_0 sends m_2 to P_1 along with its dependency vector [000001]. When P_1 receives m2, it computes its dependency vector by taking bitwise logical OR of dependency vectors of P_0 and P_1, which comes out to be [000011]. Similarly, P_2 updates its dependency vector on receiving m_3 and it comes out to be [000111]. At time t_1, P_2 initiates checkpointing algorithm with its dependency vector is [000111]. At time t_1, P_2 finds that it is transitively dependent upon P_0 and P_1. Therefore, P_2 computes the tentative minimum set [S_{minset}= {P_0, P_1, P_2}]. P_2 sends the mutable checkpoint request to P_1 and P_0 and takes its own mutable checkpoint C_{21}. For an MH the mutable checkpoint is stored on the disk of MH. It should be noted that S_{minset} is only a subset of the minimum set. When P_1 takes its mutable checkpoint C_{11}, it finds that it is dependent upon P_3 due to m_4, but P_3 is not a member of S_{minset}; therefore, P_1 sends mutable checkpoint request to P_3. Consequently, P_3 takes its mutable checkpoint C_{31}.

After taking its mutable checkpoint C_{21}, P_2 generates m_8 for P_3. As P_2 has already taken its mutable checkpoint for the current initiation and it has not received the tentative checkpoint request from the initiator; therefore P_2 buffers m_8 on its local disk. We define this duration as the uncertainty period of a process during which a process is not allowed to send any massage. The massages generated for sending are buffered at the local disk of the sender's process. P_2 can sends m_8 only after getting tentative checkpoint request or abort massages from the initiator process. Similarly, after taking its mutable checkpoint P_0 buffers m_{10} for its uncertainty period. It should be noted that P_1 receives m_{10} only after taking its mutable checkpoint. Similarly, P_3 receives m_8 only after taking its mutable checkpoint C_{31}.A process receives all the massages during its uncertainty period for example P_3 receives m_{11}. A process is also allowed to perform its normal computations during its uncertainty period.

At time t_2, P_2 receives responses to mutable checkpoints requests from all process in the minimum set (not shown in the Figure 2) and finds that they have taken their mutable checkpoints successfully, therefore, P_2 issues tentative checkpoint request to all processes. On getting tentative checkpoint request, processess in the minimum set [P_0, P_1, P_2, P_3] convert their mutable checkpoints into tentative ones and send the response to initiator process P_2; these process also send the massages, buffered at their local disks, to the destination processes For example, P_0 sends m_{10} to P_1 after getting tentative checkpoint request [not shown in the figure]. Similarly, P_2 sends m_8 to P_3 after getting tentative checkpoint request. At time t_3, P_2 receives responses from the process in minimum set [not shown in the figure] and finds that they have taken their tentative checkpoints successfully, therefore, P_2 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.

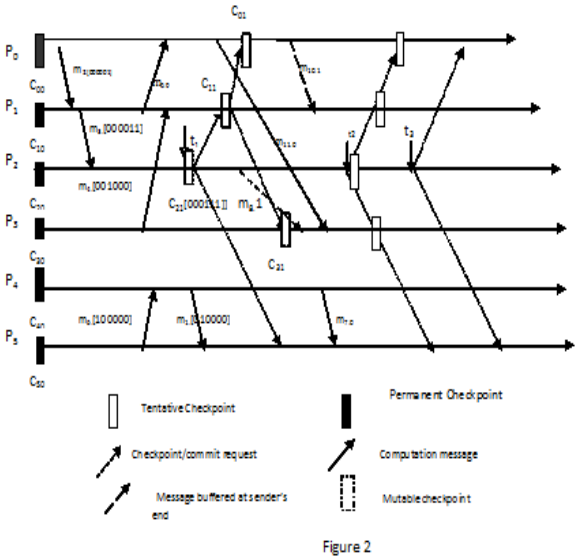

Figure 2

3. CONCLUSION

In this paper, we have proposed a minimum-process checkpointing protocol for deterministic mobile distributed systems, where no useless checkpoints are taken and an effort has been made to minimize the blocking of

processes. We try to reduce the checkpointing time and blocking time of processes by limiting checkpointing tree which may be formed in other algorithms [4, 9]. 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 try to reduce the loss of checkpointing effort when any process fails to take its checkpoint in coordination with others.

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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

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

[20] 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.

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

[22] Lalit Kumar Awasthi, Parveen 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.

[23] 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).

[24] A. Tanenbaum and M. Van Steen, Distributed Systems: Principles and paradigms,Upper Saddle River, NJ, Prentice-Hall,2003.

[25] M. Singhal and N. Shivaratri, Advanced Concepts in Operating Systems, New York, McGraw Hill, 1994.