A Fast and Efficient Non-Blocking Coordinated Movement-Based Check pointing Approach for Distributed Systems

Jayanta Datta[1]
Department of Computer Application
RCC Institute of Information technology
Kolkata-700015, West Bengal, India
***Harinandan Tunga[2]**
Department of Computer Sc. & Engineering
RCC Institute of Information technology
Kolkata-700015, West Bengal, India
Rudranath Mitra[3]
Department of Information Technology
Heritage Institute of Technology
Anandapur, Kolkata-700107, West Bengal, India

Abstract
In this paper, we have presented an efficient non-blocking coordinated check pointing algorithm for distributed systems. It produces a consistent set of checkpoints, without the overhead of taking temporary checkpoints; the algorithm also makes sure that only few processes are required to take checkpoints in its any execution; it uses very few control messages and the participating processes are interrupted fewer number of times when compared to some noted related works. The two most important criteria are non-blocking and minimum number of checkpoints. Cao-Singhal showed in their algorithm that it is impossible to design minimum process non-blocking algorithm but it is not desirable in mobile environment that underlying computation will be blocked whenever a check pointing algorithm invoked. If the check pointing scheme is blocking then the performance of the system will be highly affected by the frequent initiation of check pointing algorithm. We must try to minimize the blocking time while keeping the number of checkpoints minimum. So, the proposed scheme concentrate to minimize this overhead by combining coordinated check pointing with minimum blocking time.

Keywords: Check-pointing, Dependency Vector (DV), distributed algorithm, Mobile Support Stations (MSS), Message Handling System (MHS), and Received Pronunciation (RP).

Introduction
Distributed system consists of only static hosts. But nowadays, the needs of mobile devices have been increased greatly which in turn, gave rise of a new technology, called mobile computing. We can consider mobile computing is a special case of distributed computing system. The term 'mobile' implies able to move while retaining its network connection and a host which can move is called mobile host (MH). The infrastructure machines that can communicate directly with mobile hosts are called mobile support stations (MSS). Due to mobility and portability of devices mobile computing is characterized by some constraints [3]:

- Mobile elements are resource-poor relative to static **-** For a given cost and level of technology, considerations of weight, power, and size ergonomics will exact a penalty in computational resources such as processor speed, memory size, and disk capacity. While mobile elements will improve in absolute ability, they will always be resource**-** poor relative to static elements.
- Mobility is inherently hazardous- A Wall Street stockbroker is more likely to be mugged on the streets of Manhattan and have his laptop stolen than to have his workstation in a locked office be physically subverted. In addition to security concerns, portable computers are more vulnerable to loss or damage.
- Mobile connectivity is highly variable in performance and reliability - Some buildings may offer reliable, high-bandwidth wireless connectivity while others may only offer low-bandwidth connectivity. Outdoors, a mobile client may have to rely on a low-bandwidth wireless network with gaps in coverage.
- Mobile elements rely on a finite energy source **-** While battery technology will undoubtedly improve over time, the need to be sensitive to power consumption will not diminish. Concern for power consumption must span many levels of hardware and software to be fully effective.

Fault-tolerance [4] or graceful degradation is the property that enables a system to continue operating properly in the event of the failure of some of its components. An incremental check-pointing approach introduced by Elnozahy et.al [2]. In [1] first gives the idea, how a process in a distributed system can determine a global state of the system using special marker message during computation. There are three most important parameter of coordinated check-pointing are synchronization message overhead, number of checkpoints and blocking time. The following algorithms [1][2][5][6][7][8][9] introduce the idea to minimize the overhead. Koo–Toueg[15] propose the two phase protocol that forces only a minimal number of additional processes to take checkpoints. Prakash - Singhal[5] first introduces the algorithm which makes it possible that only a minimum number of processes take checkpoint without blocking the underlying computation during check-pointing. In[8] Weigang-Susan introduce a strategy called proxy coordinator. In [13], a hybrid check-pointing protocol has been introduced that has been combined with selective sender based message logging. The idea proposed in [14] alleviates the problem of combining pessimistic message logging with uncoordinated check-pointing protocol.

Proposed Scheme

Consider a set of n processes, {P_1, P_2 ,…, P_n } involved in the execution of a distributed algorithm. Each process P_i maintains a dependency vector DV_i of size n which is initially empty and an entry $DV_i[j]$ is set to 1 when P_i receives since its last checkpoint at least one message from P_j. It is reset to 0 again when process P_i takes a checkpoint. Each process Pi maintains a checkpoint sequence number csni. This csni actually represents the current check pointing interval of process P_i. The check-pointing interval of a process denotes all the i^{th} and (i+l) Computation performed between its I checkpoint but not the checkpoint, including the i^{th} checkpoint. The csn_i is initially set to 1 and (i+l) incremented when process P_i takes a checkpoint. In this approach we assume that only one process can initiate the check pointing algorithm. This process is known as the initiator process. We define that a process P_k is dependent on another process P_r, if process P_r since its last checkpoint has received at least one application message from process P_k. In our proposed algorithm we assume primary and secondary checkpoint request exchanges between the initiator process and the rest n-1 processes. A primary checkpoint request is denoted by R_i (i = csn_j) where i is the current checkpoint sequence number of process P_j that initiates the check pointing algorithm. It is sent by the initiator process P_j to all its dependent processes asking them to take their respective checkpoints. A secondary checkpoint request denoted by R_{si} is sent from a process P_m to a process Pn which is dependent on P_m to take a checkpoint. R_{si} means to its receiver process that i is the current checkpoint sequence number of the sender process. The control message exchange is explained with an illustration shown in Figure 1. Consider a distributed system with three processes P_1, P2, and P3. We assume that P1 initiates the check pointing algorithm. To start with, P_1 takes a checkpoint and sends a primary checkpoint request to P_2, asking it to take a checkpoint as it is directly dependent on P1. P_2 takes a checkpoint after it receives the primary checkpoint request. After taking its checkpoint P_2 sends a secondary checkpoint request to P_3 as P_3 is dependent on P_2, Process P_3 then takes its checkpoint. In this work, an application message is represented by $M_{i,x}$, which means that it is the x^{th} message sent by process P_i . The checkpoint $C_{i,j}$ represents the j^{th} checkpoint taken by P_i. We have assumed that the events of taking a checkpoint and sending a checkpoint request are done atomically. Also, each process P_i piggybacks its current checkpoint sequence number with only every first outgoing application message to another process after taking We now state the situations in general when a process P_i needs to take a checkpoint. In our approach a process P_i takes a checkpoint if any of the following events occurs - if P_i is the initiator then if it receives a primary checkpoint request from the initiator. The first time it receives a secondary checkpoint request and prior to that it has not received any primary checkpoint request or any piggybacked application message. The first time it receives an application message piggybacked with the checkpoint sequence number, and prior to that it has not received any primary or secondary checkpoint request message.

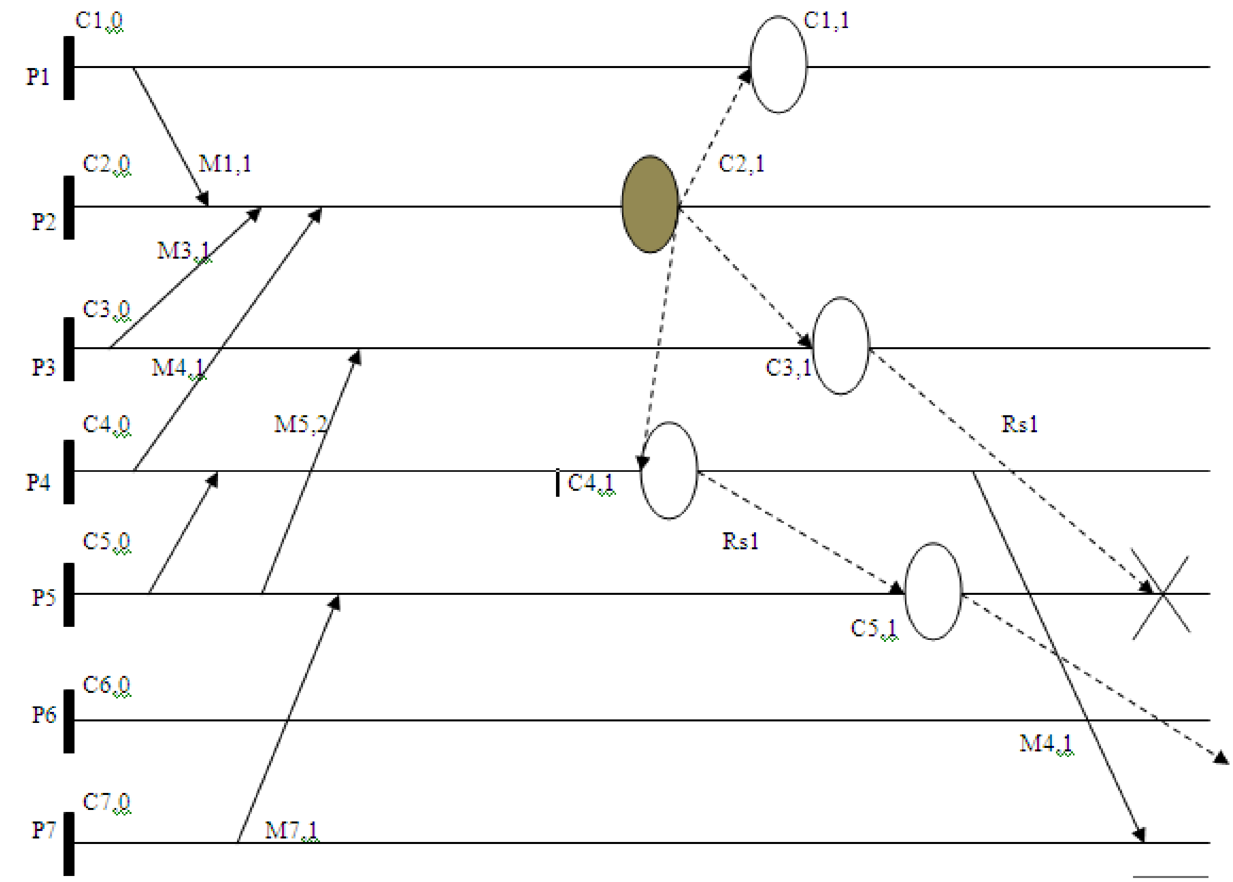

Figure 1: Process of taking Non-Blocking Coordinated checkpoint

Illustration

The behavior of each process in our approach is explained with the help of the following example. Unless otherwise mentioned a checkpoint request represents either a primary request or a secondary request. Note that an application message with piggybacked checkpoint sequence number, which may force a checkpoint to be taken at the receiving process may also be viewed as a checkpoint request. In our work a checkpoint means a permanent checkpoint. Consider the distributed system as shown in the Fig. 2. Assume that process P2 initiates the check pointing algorithm. First process P2 takes its permanent checkpoint C2,1. It then checks its dependency vector DV2[] which is {1, 0,1,1,0,0,0}. This means that process P2 has received at least one message from processes P1, P3, and P4, and since P2 has already taken its checkpoint C2,1 these messages will become orphan if P1, P3, and P4 do not take checkpoints. Therefore P2 sends primary checkpoint request R1 (csn2 = 1) to P1, P3, and P4. After sending the primary checkpoint request process P2 increments its checkpoint sequence number csn2 to 2 and finishes its participation associated with the current execution of the algorithm and continues with its normal computation. It shows the non-blocking nature of our approach. On receiving the primary checkpoint request R1 from P2, process P3 first takes a checkpoint C3,1 and then it checks its own dependency vector DV3[] which is {0,0,0,0,1,0,0}. Therefore process P3 sends a secondary checkpoint request Rs1 to process P5. Then its checkpoint sequence number csn3 is incremented to 2. Similarly processes P1 and P4 first take checkpoints C1,1 and C4,1 respectively, then each process checks its dependency vector to find the dependent processes. Process P1 finds that its dependency vector DV1 [] is null. Hence it increments its checkpoint sequence number to 2, and continues normal execution. Process P4 finds that it has received a message from process P5. Hence P4 sends a secondary checkpoint request Rs1 to process P5. It then increments its checkpoint sequence number csn4 to 2, and continues normal execution. At process P5 let us assume that the secondary checkpoint request Rs1 sent by process P4 reaches before the secondary checkpoint request sent by process P3. On receiving the secondary checkpoint request Rs1 from process P4, P5 checks its own checkpoint sequence number csn5 with that of the received checkpoint sequence number. P5 finds that its current checkpoint sequence number (csn5= 1) is not greater than the received checkpoint sequence number which is also equal to 1. Hence it decides to take a checkpoint and takes checkpoint C5,1.

After taking the checkpoint it checks its dependency vector DV5 and finds that process P7 has sent a message to it. Hence it sends a secondary checkpoint request Rs1 to P7. After sending the request it increments its checkpoint sequence number csn5 from 1 to 2. Assume that later process P5 receives the secondary checkpoint request sent by process P3. As soon as process P5 receives the checkpoint request it compares its current checkpoint sequence number csn5 with the received checkpoint sequence number. It finds that its current checkpoint sequence number (csn5 = 2) is greater than the received checkpoint sequence number which is 1. Hence it discards the checkpoint request. The above discussion takes care of the first three situations about when a process takes a checkpoint. Below, we consider the fourth situation. Suppose that process P4 after taking the checkpoint continues normal execution and sends an application message M4,1 to process P7. Since the application message is the first application message to process P7 from P4 after taking the checkpoint, it is piggybacked with the current checkpoint sequence number (csn4) of process P4 which is 2. Process P7 on receiving the application message piggybacked with the checkpoint sequence number compares its current checkpoint sequence number csn7 with the received checkpoint sequence number. It finds that the received checkpoint sequence number is equal to 2 and is greater than its current checkpoint sequence number (csn7) which is equal to 1. Therefore process P7 decides to take a checkpoint before processing the application message M4,1. P7 then takes its checkpoint C7,1 and increments its checkpoint sequence number to 2 and then processes the application message M4,1. Eventually process P7 also receives the secondary checkpoint request sent by process P5. P7 first compares its current checkpoint sequence number with the received checkpoint sequence number which is 1. It finds that its current checkpoint sequence number is greater than the received checkpoint request. Hence P7 discards the secondary checkpoint request as it has already taken its checkpoint for the current execution of the algorithm. In the above example we observe that P7 sent a message M7,1 to P5. So even if there was no such piggybacked message as M4,1, process P7 would eventually receive the secondary check-pointing request Rs,1 from P5 and take its checkpoint C7,1. Observe that because of the non-blocking nature of the algorithm the following situation may arise as well. Consider that there was no such message as M7,1; that is, assume that P7 has not sent any application message to any process at all. However, assume that it receives the piggybacked message M4,1 from P4. In our approach P7 will take its checkpoint and then process the message and then would behave like any other process involved in the check pointing approach.

Data Structures

- Status: A Boolean variable maintained at each process P_i. If $Status_i$ =1, then P_i is in a check-pointing phase. When P_i receives a checkpoint request it sets $Status_i$ =1 and after receiving the checkpoint commit message it resets $Status_i$ =0.
- DP: A Boolean array of size n, maintained by MSS on behalf of its local MHs. $DP_i[j]$=1 means process P_i receives some computation messages from P_j. All elements of this array are initialized to 0 except $DP_i[i]$=1.
- RP: A Boolean array of size n, maintained by MSS on behalf of its local MHs. It is used to save dependency relation during the check-pointing interval. It is same as DP. After that interval n-bitwise OR operation is performed between the elements of DP and RP and the result is stored to DP. Then RP is refreshed.
- Count: It is an integer variable stored at process P_i. It is initialized to 0. Each time the check-pointing algorithm invoked by P_i, $count_i$ is incremented by 1.
- Mark: It is a Boolean variable which is used to indicate the blocking period at the receiver side MSS. If mark = 1, that means the MSS is waiting for the final dependency list and that time all incoming messages will be buffered at MSS.

Assumption

1. Processes communicate only through messages. They do not share any common memory or common clock
2. All communication channels between MHs and MSSs are FIFO. The channels between MSS and MSS are also FIFO.
3. No process fails during the check pointing phase.
4. Channels are lossless. Messages arrive with an arbitrary but finite delay.
5. A process will not receive any checkpoint request from another initiator before the current executing one is completed.

First Phase

1. When process P_i running on MH_i wants to save its state, it takes a tentative checkpoint and informs its current MSS, MSS_p so that MSS_p starts the checkpointing algorithm as a proxy coordinator on behalf of Pi. MSSp sets $mark_p = 1$.

2. MSSp sends request to all other MSSs in the system to collect the dependency vectors of other MHs in the system.

3. All other MSSs in the system respond to the request by sending the dependency vectors of their local MHs and starts waiting for final dependent set. After sending the dependency sets of their MHs, MSS_q (q!=p) sets *markq* = 1

3.1 MSS_q receives a computation message for a process P_j which is currently under it, if $mark_q = 1$, then MSS_q buffers the message and update the dependency information in RP_j[].

3.2 MSS_q receives an outgoing computation message from an MH currently under it. It doesn't block the message. It forwards the message to the MSS of the receiver process.

3.3 If MSS_p receives a message, it checks the value of *mark* and if *mark* is set to 1, then it checks whether the receiver is the initiator i.e. MH_i. If receiver = MH_i then it forwards the message, otherwise buffers it.

4. MSS_p, the proxy coordinator receives the dependency vectors from other MSSs. After that MSS_p constructs an NxN dependency matrix with one row per process, represented by the dependency vectors of the process. Based on this NxN matrix, MSS_p can locally calculates both (direct and transitive) dependents of P_i.

5. MSS_p broadcasts the final dependent list to all other MSSs.

6. On receipt of the dependent list, MSS_q checks the buffered messages.

6.1 If receiver process (i.e. the process belongs to MSS_q) of the buffered message is in the dependent list, then MSS_q attaches a flag=1 with that message and sends to the intended process.

6.2 If sender process of the buffered message is in the dependent list, then MSS_q keeps the payload of the message in stable log on behalf of the receiving process and sends the message to the intended process.

6.3 If both the sender and the receiver is in the dependent list, then MSS_q checks the *status* of the sender.

6.3.1 If *status* of sender = 1, then set flag = 1.

6.3.2 If *status* of sender = 0, just delivers the message as it is.

7. MSS at the receiver side keeps the copy of the message in its volatile log. When MSS_q receives the final dependency list, it which MHs within its area are not in the dependency list. Then MSS_q checks the dependency list of process P_k, which is not in the final set. If MSS_q finds $DP_k[j] = 1$ and P_j belongs to final dependent set. Then MSS_q finds all the messages with sender P_j from the temporary log of P_k and flushed them to the stable storage.

8. When all buffered messages have been delivered MSS_q resets $mark_q = 0$. sends checkpoint request to all its local processes which are in the dependent list.

9. Process P_j receives a computation message and it checks the value of the flag bit attached to the message

   9.1 If flag =1, then P_j takes a tentative checkpoint and sets its *status* to 1. Then it processes the message.
   9.2 If flag =0, then P_j simply processes the message.

10. Process P_j receives a checkpoint request message. It checks the value of $Status_j$. If $Status_j = 1$, it discards the checkpoint request and if $Status_j = 0$, it takes a tentative checkpoint, sets its *status* to 1 and sends back a reply to its MSS_q.

11. MSS_q has received reply messages from all the processes to which it sent checkpoint request messages, it sends a reply message to the initiator, MSS_p.

Second Phase

1. If MSS_p receives reply from all the dependent processes, then it broadcast a commit message to all the MSSs in the system. Otherwise abort the check-pointing algorithm.

2. On receipt of the commit message, the tentative checkpoint becomes permanent. The elements of dependency vectors DP of the processes, which have taken checkpoint, are refreshed and elements of RP are copied to DP.

Optimization

The performance of a check-pointing algorithm is determined by three parameters - blocking time, synchronization message overhead and number of checkpoints required. N= Number of MHs and M= Number of MSSs and N >> M. Let us assume, all processes running on the MHs and there is only one process running on each MH.

Experimental Results

Algorithm	Blocking Time	Messages
Koo-Toueg[8]	$N^* (4^* T_{mh} + T_{chkpt} + T_{search})$	$N^*(6^* C_{mh} + C_{search})$
Cao-Singhal[11]	$2^* T_{mss}$	$3N^* C_{mh}$
Proposed Scheme	T_{mss}	$3N^* C_{mh}$

Table 1: Comparative Study of Proposed Scheme

Meaning of Notations

- C_{mss} = cost of sending message between any two MSSs.
- C_{mh} = cost of sending a message from an MH to its local MSS.
- C_{broad} = cost of broadcasting a message in the static network.
- C_{search} = cost incurred to locate an MH and forward amessage to its current local MSS, from a source MSS.
- T_{mss} = average message delay in the static network.
- T_{mh} = average message delay in the wireless network.
- T_{search} = average delay incurred to locate an MH and forward a message to that MH.
- T_{chkpt} = Average delay to save a checkpointon the stable storage

In order to measure buffering time - after an MSS has sent all its local dependent vectors to the proxy MSS; it can't forward any computation message to its local MH until it receives the final set of dependent processes. An MSS buffers the messages during this time. Total blocking time of Cao-Singhal Algorithm was $2T_{mss.}$ When an MSS of a sending process receives a computation message immediate after sending the dependency vector to the proxy MSS, it forwards the message to the MSS of receiving process and the message will be buffered their. So, in worst case, the maximum time of buffering will be $T_{buffer} = 2T_{mss} - T_{search}$.

So, $T_{buffer} <= T_{mss.}$ Since $T_{search} >= T_{mss}$

Therefore, the computation will not be blocked for $2T_{mss}$ time.

Handling Lost Message- In this scheme, both the send and receive event is recorded in the dependency vector. So, here the lost messages have been handled properly. In figure 4.1, messages m_5 and m_1 will be lost though they have reached and processed successfully by process P_2 and $P_{6,}$ if only receive events are stored in the dependency vector. Because sending event of messages m_5 and m_1 are recorded since process $P_{3,}$ $P_{4,}$ P_5 will take checkpoints in case of storing only receive events. But here, both send and receive have been recorded in global checkpoint. No broadcast message - in this scheme, there is a broadcast message at the MH level. So, it will not interrupt that process that is in doze mode and hence, fulfilling the limited battery power constraint of mobile hosts. Search Cost-The mobility of hosts in mobile computing environment incurs a large amount of search delay and hence search cost. In this scheme, there is no search cost for checkpoint request messages since initiator broadcasts the final set of dependents to every MSS in the system. The MSSs forward the request message only to their local MHs.

Conclusion

The proposed scheme is developed to reduce the blocking time of the coordinated check pointing algorithm. The above scheme might be more optimized in terms of blocking time. The recovery issue has not been discussed here. Further developments are being carried out to handle recovery issues.

Acknowledgements

We express our sincere gratitude to seiner faculties, Department of Computer Science, Information Technology and Engineering and Computer Application, RCC Institute of Information Technology and Heritage Institute of Technology for extending his valuable time for us to take up this problem as a paper and also financial support our institutes. This paper deals with a system that to the best of our belief does not exist as yet.

Reference

[1] M. Chandy and L. Lamport. “Distributed snapshots: Determining global states of distributed systems.” In ACM Transactions on Computing Systems, 3(1):63— 75, Aug. 1985.

[2] E.N. Elnozahy, D.B. Johnson, and W. Zwaenepoel. “The performance of consistent checkpointing.” In Proceedings of the Eleventh Symposium on Reliable Distributed Systems, pp. 39— 47, Oct. 1992.

[3] B. Randell. “System structure for software fault-tolerance.” IEEE Transactions on Software Engineering, SE-1(2):220—232, Jun. 1975.

[4] A Clematis. “Fault-tolerant programming for network based parallel computing.” In Microprocessing and Microprogramming, vol. 40, pp. 765— 768, 1994.

[5] Prakash, R. and M. Singhal, “Low-Cost Checkpointing and Failure Recovery in Mobile Computing Systems,” IEEE Trans. Parallel and Distributed Systems, pp.1035-1048, Oct. 1996.

[6] Cao, G. and M. Singhal, “On Coordinated Checkpointing in Distributed Systems,” IEEE Trans. Parallel and Distributed Systems, pp. 1213-1225, Dec. 1998.

[7] Guohong Cao and Mukesh Singhal “On the Impossibility of Min-Process Non-Blocking Checkpointing and An Efficient Checkpointing Algorithm for Mobile Computing Systems” Department of Computer and Information Science The Ohio State University Columbus, OH 43210

[8] Weigang Ni, Susan V. Vrbsky and Sibabrata Ray “Low-cost Coordinated Nonblocking Checkpointing in Mobile Computing Systems”, Department of Computer Science University of Alabama Tuscaloosa, AL 35487-0290, Proceedings of the Eighth IEEE International.

[9] Robert H. B. Netzer and Jian Xu “Necessary and Sufficient Conditions for Consistent Global Snapshots”IEEE transactions on parallel and distributed systems, vol. 6, no. 2, february 1995

[10] Guohui Li and LihChyun Shu “A Low-Latency Checkpointing Scheme for Mobile Computing Systems” Processing of the IEEE 29th Annual International Computer Software and Application Conference (COMPSAC’05), 2005.

[11] Y. M. Wang. “Space reclamation for uncoordinated checkpointing in message-passing systems.” Ph.D. Thesis, University of Illinois Urbana-Champaign, Aug. 1993.

[12] Y. M. Wang. “Consistent global checkpoints that contain a set of local checkpoints.” IEEE Transactions on Computers, 46(4):456— 468, Apr. 1997.

[13] Kwang-Sik Chung, Ki-Bom Kim, Chong-Sun Hwang, Jin Gon Shon and Heon-Chang Yu “Hybrid Checkpointing Protocol Based on Selective Sender-based Message Logging”. IEEE , 1997.

[14] Mehdi Aminian, Mohammad k. Akbari and Bahman Javadi “Combining Coordinated and Uncoordinated Checkpointing with Pessimistic Message Logging”. IJCSNS International Journal of Computer Science and Network Security, Vol. 6 NO. 4, 2006.

[15] 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.