Volume 2, No. 2, Mar-Apr 2011

ISSN No. 0976-5697

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

Efficient Coordinated Checkpointing Protocol for Mobile Distributed Systems

Mukesh kumar*
Research Scholar, Department of CSE
Singhania University,
Pacheri Bari ,Rajasthan ,India
mrana91@gmail.com

Parveen Kumar
Professor
MIET, Meerut ,India
pk223475@yahoo.com

***Abstract*:** Mobile Computing represents a new paradigm that aims to provide continuous network connectivity to users regardless of their location. A wide spectrum of portable, personal computing devices has recently been introduced in the market that range from laptop computers to handheld personal digital assistants. Coupled with the advent of wireless networking, this has given rise to a new style of computing wherein the computer can move with the user and yet maintain its network connections. Solutions to communication and synchronization problems in distributed systems have so far been designed for networks comprising solely of static hosts.

Keywords: Distributed system, mobile computing, consistent global state, coordinated checkpointing systems.

1. INTRODUCTION

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 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.

Synchronous 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 Pi takes its checkpoint only if it a member of the minimum set (a subset of interacting process). A process Pi is in the minimum set only if the checkpoint initiator process is transitively dependent upon it. Pj is directly dependent upon Pk only if there exists m such that Pj receives m from Pk in the current checkpointing interval [CI] and Pk has not taken its permanent checkpoint after sending m. The ith CI of a process denotes all the computation performed between its ith and (i+1)th checkpoint, including the ith 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.

II. SYSTEM MODEL

There are n spatially separated sequential processes P0, P1,.., Pn-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. 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.

3. DATA STRUCTURE

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.

(i) ***Each process Pi maintains the following data structures, which are preferably stored on local MSS:***

p-csn$_i$:	A monotonically increasing integer checkpoint sequence number for each process. It is incremented by 1 on mutable checkpoint.
tentative$_i$	A flag that indicates that Pi has taken its tentative checkpoint for the current initiation.
dd_set$_i$[]:	A bit vector of size n; dd_seti[j] is set to '1' if Pi receives a message from Pj such that Pi becomes directly dependent upon Pj for the current CI. Initially, the bit vector is initialized to zeroes for all processes except for itself, which is initialized to '1'. For MHi it is kept at local MSS. On global commit, dd_set [] of all processes are updated. In all-process checkpointing, each process initializes its dd_set [] on tentative checkpoint.
blocking$_i$	A flag that indicates that the process is in blocking period. Set to '1' when Pi receives the dd_set [] request; set to '0' on the receipt of mutable checkpoint request for a non-minimum-set process. 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.
buffer$_i$:	A flag. Set to '1' when Pi buffers first message in its blocking period.
C_state$_i$	A flag. Set to '1' on the receipt of the minimum set. Set to '0' on receiving commit or abort.

(ii***) Initiator MSS maintains the following Data structures:***

min_vect[]:	A bit vector of size n. Computed by taking transitive closure of dd_set [] of all processes with the dd_set [] of the initiator process [4]. Minimum set={Pk such that min_vect [k]=1}.
R_tent[]:	A bit vector of length n. r_tent[i] is set to '1' if Pi has taken a tentative checkpoint.
R_mut[]:	A bit vector of length n. r_mut[i] is set to '1' if Pi has taken a mutable checkpoint.
Timer1:	A flag; set to '1' when maximum allowable time for collecting minimum-process global checkpoint expires.

(iii) Each MSS (including initiator MSS) maintains the following data structures:

D[]:	A bit vector of length n. D[i]=1 implies Pi is running in the cell of MSS.
Ee_tent[]:	A bit vector of length n. EE_tent[i] is set to '1' if Pi has taken its tentative Checkpoint.
Ee_mut[]:	A bit vector of length n. EE_mut[i] is set to '1' if Pi has taken a mutable checkpoint.
→_bit:	A flag at MSS. Initialized to '0'. Set to '1' when some relevant process in its cell fails to take its tentative checkpoint.
P$_{in}$:	Initiator process identification.
csn[]	An array of size n, maintained on every MSS, for n processes. csn[i] represens the most recently committed checkpoint sequence number of Pi. 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.
G_chkpt:	A flag which is set to '1' on the receipt of (i) checkpoint request in all-process checkpointing or (ii) dd_set [] request in minimum-process algorithm.
Chkpt	A flag which is set to 1 when the MSS receives the checkpoint request in the minimum-process algorithm.
mss_id	An integer. It is unique to each MSS and cannot be null.

IV. DESIGN OF COORDINATED CHECK-POINTING PROTOCOL

The mechanism of our protocol is following: 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_ida) 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_idb) and mss_ida is lower than mss_idb,the, current initiation with mss_ida is discarded and the new one having mss_idb 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 MSSj, takes the following actions. It sends the mutable checkpoint request to Pi only if Pi belongs to the min_vect [] and Pi is running in its cell. On receiving the checkpoint request, Pi takes its mutable checkpoint and informs MSSj. On receiving positive response from Pi, MSSj updates p-csni, resets blocking, and sends the buffered messages to Pi, if any. Alternatively, If Pi is not in the min_vect [] and Pi is in the cell of MSSj, MSSj resets blockingi and sends the buffered message to Pi, 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, Pi processes m, received from Pj, if following conditions are met: (i) (!buferi) i.e. Pi has not buffered any message (ii) (m.psn <=csn[j]) i.e. Pj has not taken its checkpoint before sending m (iii) (dd_seti[j]=1) Pi is already dependent upon Pj in the current CI or Pj has taken some permanent checkpoint after sending m. Otherwise, the local MSS of Pi buffers m for the blocking period of Pi 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.

A. Example

We explain the proposed minimum-process checkpointing algorithm with the help of an example. In Figure 1, at time t1, P4 initiates checkpointing process and sends request to all processes for their dependency vectors. At time t2, P4 receives the dependency vectors from all processes (not shown in the Figure 1) and computes the minimum set (min_vect[]) which is {P3, P4, P5}. P4 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 P3 and P5 get the min_vect[], they find themselves in the min_vect[]; therefore, they take their mutable checkpoints. When P0, P1 and P2 get the min_vect [], they find that they do not belong to min_vect [], therefore, they do not take their mutable checkpoints

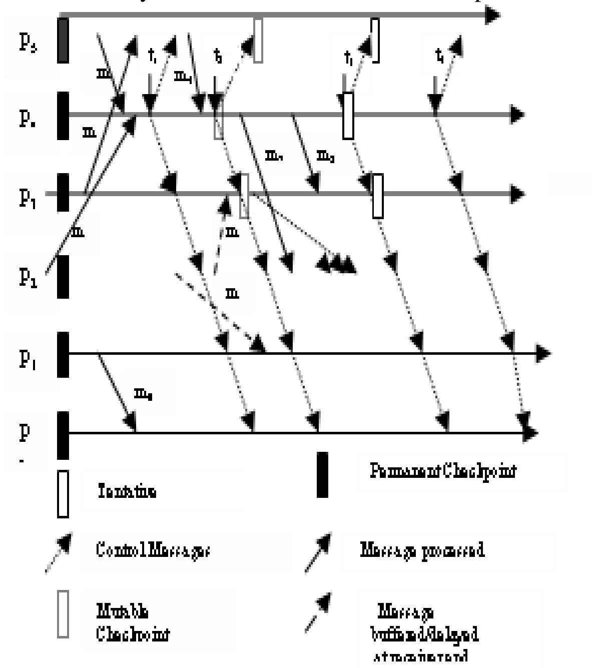

Figure 1: Example of Proposed Protocol

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. P4 receives m4 during its blocking period. As dd_set4[5]=1 due to m3, and receive of m4 will not alter dd_set4[]; therefore P4 processes m4. P1 receives m5 from P2 during its blocking period; dd_set1[2]=0 and the receive of m5 can alter dd_set1[]; therefore, P1 buffers m5. Similarly, P3 buffers m6. P3 processes m6 only after taking its mutable checkpoint. P1 process m5 after getting the min_vect [].P2 processes m7 because at this movement it not in the blocking state. Similarly, P3 processes m8. At time t3, P4 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 t4, P4 receives responses to tentative checkpoint requests from all relevant processes (not shown in the Figure 1) and issues the commit request.

V. CONCLUSION

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.

VI. 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 Ensures Rollback-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.