ISSN 2278-3091
Volume 10, No.4, July - August 2021
International Journal of Advanced Trends in Computer Science and Engineering
Available Online at http://www.warse.org/IJATCSE/static/pdf/file/ijatcse321042021.pdf
https://doi.org/10.30534/ijatcse/2021/321042021

Transient-Snapshot based Minimum-process Synchronized Check pointing Etiquette for Mobile Distributed Systems

Deepak Chandra Uprety[1], **Dr. Parveen Kumar**[2], **Dr. Arun Kumar Choudhary**[3]
[1] Research Scholar, Department of Computer Science Engg, Nims University, Jaipur (Raj), deepak.glb@gmail.com
[2] Professor, Department of Computer Science Engg., Nims University, Jaipur (Raj), parveen.kumar@nimsuniversity.org,
[3] Professor, AP Goyal Shimla University, Shimla, Himachal Pradesh, India, choudharyarun@rediffmail.com

ABSTRACT

Minimum-process harmonized checkpointing is well thought-out an attractive methodology to acquaint with fault tolerance in mobile systems patently. We design a minimum-process synchronous checkpointing algorithm for mobile distributed system. We try to minimize the intrusion of processes during checkpointing. We collect the transitive dependencies in the beginning, and therefore, the obstructive time of processes is bare minimum. During obstructive period, processes can do their normal computations, send messages and can process selective messages. In case of failure during checkpointing, all applicable processes are necessitated to abandon their transient snapshots only. In this way, we try to reduce the loss of checkpointing effort when any process fails to take its checkpoint in coordination with others. We also try to minimize the harmonization message complexity during checkpointing.

Key words: Mobile Computing Systems, coordinated checkpointing, Recovery.

1. INTRODUCTION

In mobile distributed computing Systems, some methods are functioning on mobile nodes (Mob_Nodes). A Mob_Node is a computer that may retain its connectivity with the rest of the distributed frame of reference through a wireless network while on move; or it may detach. It necessitates assimilation of portable computers within existing data network. A Mob_Node can join to the network from diverse sites at dissimilar times. The groundwork mechanisms that interconnect directly with the Mob-Hosts are called Mobile Support Stations (M_S_Sts). A cubicle is a logical or topographical exposure area under an M_S_St [9, 19, 20].

Local reinstatement_point is the hoarded state of a method at a processor at a given instance. Global snapshot is an assortment of local reinstatement_points, one from each method. A global state is said to be "consistent" if it contains no orphan application_communication; i.e., an application_communication whose receive event is documented, but it sends event is vanished. To recuperate from a catastrophe, the system resurrects its accomplishment from a preceding CGS (Consistent Global State) saved on the stable storage during fault-free accomplishment. This saves all the computation done up to the last CGS and only the working out done subsequently, prerequisites to be recreated.Processes in a distributed frame of reference communicate bysending and receiving communications [1, 7, 14, 17, 18].

Checkpointing / CGS_assortment (Consistent Global State assortment) for Mobile_DS (Mobile Distributed Systems) needs to handle new issues like: mobility, low bandwidth of wireless channels, lack of stable storage on mobile nodes, disconnections, limited battery power and high failure rate of mobile nodes. These concerns make customary CGS_assortment procedures inappropriate for such settings. least_int_method (least interacting method) collaborative CGS_assortment is an appropriate methodology to acquaint with fault tolerance in Mobile_DS patently. This approach is domino-free, requires at most two recovery-points of a method on established storage, and necessitates only a least number of methods to capture snapshots. But it requires extra orchestration communications, hindering of the underlying working out or taking some unserviceable recovery_points [3, 4, 5, 6, 12, 13, 15, and 16]**.**

In this paper, we put forward a least_int_method collaborative CGS_assortment etiquette for non-deterministic Mobile_DS, where no unserviceable reinstatement_points are captured. We use the technique to minimize the hindering of methods. During the period, when a method sends its causal_depend_array (causal dependency array) to the originator and receives the least_int_method_set[], may receive some application_communications, which may add new members to the already computed least_int_method_set[]. Such application_communications are buffered at the receiver side. It should be noted that the duration for which the application_communications are delayed at the receiver's end is insignificantly small.

We also try to curtail the loss of CGS_assortment effort when any method miscarries to register its reinstatement_point in harmonization with others. We suggest that in the first phase, all pertinent Mob_Nodes will register transient reinstatement_point only. Transient reinstatement_point is stored on the memory of Mob_Node only. In this case, if some method miscarries to register its reinstatement_point in the first phase, then Mob_Nodes need to abandon their transient reinstatement_points only. The effort of taking a transient reinstatement_point is trivial as paralleled to the tentative one. We put forward three phase etiquettes for CGS_assortment. But, in the suggested etiquette, the harmonization with the originator M_S_St is done without sending explicit orchestration communications. We want to emphasize that in all collaborative CGS_assortment schemes, available in literature, harmonization among methods and originator takes place by directing categorical orchestration communications [2, 3, 4, 7]. In this way, we try to significantly diminish the orchestration overhead in collaborative CGS_assortment.

In order to keep the hindering of methods bare minimum, we assemble the causal_depend_arrays[] (causal dependency arrays) and compute the exact least_int_method_set[] in the beginning of the etiquette as in [3]. The number of methods that register reinstatement_points is curtailed to 1) avoid arising of Mob_Nodes in doze mode of operation, 2) curtail whipping of Mob_Nodes with CGS_assortment action, 3) save limited battery life of Mob_Nodes and low bandwidth of wireless channels.

The new ideas used in this etiquette are given as follows. In the suggested etiquette, the harmonization with the originator M_S_St is done without sending explicit orchestration communications. The originator M_S_St (say M_S_St$_{in}$) collects the causal_depend_array [] of all methods, computes the least_int_method_set [] and broadcasts the transient reinstatement_point invitation to all M_S_Sts along with the least_int_method_set[] . Suppose, M_S_St$_i$ gets the transient reinstatement_point invitation in the first phase from M_S_St$_{in}$. It sets its timer (timer_transient) and sends the transient reinstatement_point invitation to all pertinent local Mob_Nodes. timer_transient is the extreme permissible time for all pertinent methods to register their transient reinstatement_points. On receiving the transient reinstatement_point invitation, a Mob_Node registers its transient reinstatement_point and sends the response to M_S_St$_i$. Before the expiry of the timer_transient, if M_S_St$_i$gets the negative response from some Mob_Node to its transient reinstatement_point invitation, then M_S_St$_i$ sends the negative response to M_S_St$_{in}$; and M_S_St$_{in}$ issues abandon communication to all M_S_Sts. Otherwise, on expiry of timer_transient, if M_S_St$_i$ does not get the positive response to transient reinstatement_point invitation from all pertinent local Mob_Nodes, it informs failure communication to M_S_St$_{in}$and M_S_St$_{in}$issues abandon broadcast. Alternatively, on expiry of timer_transient, M_S_St$_i$ issues tentative reinstatement_point invitation to the pertinent Mob_Nodes in its cubicle and sets timer_tent. On expiry of timer_transient, if M_S_St$_i$ does not get abort massage from M_S_St$_{in}$, it is presumed that all pertinent methods have captured their transient reinstatement_points; and the etiquette should enter the second phase in which all pertinent methods convert their transient reinstatement_points into the tentative ones. Similarly, timer_tent is the maximum allowable time for all pertinent methods to convert their transient reinstatement_points into tentative ones. If some method fails to register its tentative reinstatement_point, then M_S_St$_i$ informs M_S_St$_{in}$and M_S_St$_{in}$ issues abort. Otherwise, after the timeout of timer_tent, M_S_St$_i$ commits the reinstatement_points of the methods of the least_int_method_set [], which are local to its cubicle. On expiry of timer_tent, if M_S_St$_i$ does not get abort massage from M_S_St$_{in}$, it is presumed that all pertinent methods have captured their tentative reinstatement_points; and the etiquette should enter the third phase in which all pertinent methods convert their tentative reinstatement_points into the permanent ones. In this way, three-phase collaborative CGS_assortment etiquette commits without sending or receiving much orchestration communications. Only in the case of a failure, an M_S_St issues the failure communication to M_S_St$_{in}$ and M_S_St$_{in}$ issues the abandon. The suggested etiquette may register longer time to commit. But in doing so, we are saving orchestration communications to significant extent and no extra hindering of methods takes place due to longer commit time.

2. THE PROPOSED CHECKPOINTING ALGORITHM

2.1 System Model and Data Structures

Our frame of reference model is similar to [4]. The list of data structures is given as follows. All data structures are adjusted on accomplishment of a CGS_assortment method, if not mentioned unambiguously.

(a) Each method *P_i* maintains the following data structures, which are preferably stored on local M_S_St:

p-c_s_n$_i$
A monotonically increasing integerreinstatement_point sequence number for each method. It is incremented by 1 on transient reinstatement_point.

tentative$_i$
A flag that indicates that P_i has captured its tentative reinstatement_point for the current initiation.

cdd_set []
A bit array of size *n; cdd_set$_i$ [j]* is set to '1' if P_i receives an application_communication from P_j such that P_i becomes causally dependent upon P_j for the current CI. Initially, the bit array is initialized to zeroes for all methods except for itself, which is initialized to '1'. For Mob_Node$_i$ it is kept at local M_S_St. On global commit, *cdd_set [] of* all methods are updated.

hindering$_i$
A flag that indicates that the method is in hindering period. Set to '1' when P_i receives the *cdd_set []* invitation; A method comes out of the hindering state only after taking its transient instatement_point if it is a member of the least_int_method_set []; otherwise, it comes out of hindering state after getting the transient reinstatement_point invitation.

$Buffer_i$

A flag. Set to ‘1’ when P_i buffers first application_communication in its hindering period.

c_state_i

A flag. Set to ‘1’ on the receipt of the least_int_method. Set to ‘0’ on receiving *commit* or *abort*.

(b) Initiator M_S_St maintains the following Data structures

least_int_method_set []

A bit array of size *n*. Computed by taking transitive closure of *cdd_set []* of all methods with the *cdd_set []* of the originator method. Minimum set= {P_k such that *least_int_method_set [k]* =1}.

r_tent []

A bit array of length *n. r_tent [i]* is set to ‘1’ if P_i has captured a tentative reinstatement_point.

r_mut []

A bit array of length *n*. *r_mut [i]* is set to ‘1’ if P_i has captured a transient reinstatement_point.

timer1

A flag; set to ‘1’ when maximum allowable time for collecting least_int_method global reinstatement_point expires.

(c) Each M_S_St (including originator_M_S_St) maintains the following data structures

D []

A bit array of length n. *D[i]* =1 implies P_i is running in the cubicle of M_S_St.

ee_tent []

A bit array of length *n. EE_tent[i]* is set to ‘1’ if P_i has captured its tentative reinstatement point.

ee_mut []

A bit array of length *n. EE_mut [i]* is set to ‘1’ if P_i has captured a transient reinstatement point.

s_bit

A flag at M_S_St. Initialized to ‘0’. Set to ‘1’ when some relevant method in its cubicle fails to register its tentative reinstatement_point.

P_{in}

Initiator method identification.

c_s_n []

An array of size n, maintained on every M_S_S, for n methods. c_s_n[i] represents the most recently committed reinstatement_point sequence number of P_i. After the commit operation, if m_vect[i] =1 then c_s_n[i] is incremented. It should be noted that entries in this array are updated only after converting tentative reinstatement_points in to permanent reinstatement_points and not after taking tentative reinstatement_points.

G_chkpt

A flag which is set to ‘1’ on the receipt of (i) reinstatement_point invitation in all-method CGS_assortment or (ii) *cdd_set []* invitation in least_int_method etiquette.

Chkpt

A flag which is set to 1 when the M_S_St receives the reinstatement_point invitation in the least_int_method etiquette.

Mss_id

An integer. It is unique to each M_S_St and cannot be null.

timer_transient

It shows the maximum allowable time for all pertinent methods to register their transient reinstatement_points. It also includes the time in which an M_S_St informs the $M\_S\_St_{in}$ and $M_S_St_{in}$informs all M_S_Sts.

timer_tent

It shows the maximum allowable time for all pertinent methods to convert their transient reinstatement_points into tentative ones. It also includes the time in which an M_S_St informs the $M\_S\_St_{in}$ and $M_S_St_{in}$informs all M_S_Sts.

2.2 Proposed Algorithm

The originator M_S_St newscasts an invitation to all M_S_Sts to send the *cdd_set []* arrays of the methods in their cubicles. All *cdd_set []* arrays are at M_S_Sts and thus no initial CGS_assortment control_communications or responses travel wireless channels. On receiving the *cdd_set []* invitation, an M_S_St records the identity of the originator M_S_St (say mss_id_a) and M_S_St, sends back the *cdd_set []* of the methods in its cubicle, and sets *g_chkpt*. If the originator_M_S_St receives an invitation for *cdd_set []* from some other M_S_St (say mss_id_b) and mss_id_a is lower than mss_id_b, then, current initiation with mss_id_a is rejected and the new one having mss_id_b is sustained. Correspondingly, if an M_S_St receives *cdd_set []* invitations from two M_S_Sts, then it discards the invitation of the originator M_S_St with lower mss_id. Otherwise, on receiving *cdd_set []* arrays of all methods, the originator_M_S_St *computes least_int_method_set [], sends* transient reinstatement_point invitation along with the *least_int_method_set []* to all M_S_Sts. In this way, if two methods contemporaneously start CGS_assortment, then one is snubbed. When a method sends its *cdd_set []* to the originator M_S_St, it comes into its hindering state. A method comes out of the hindering state only after taking its transient reinstatement_point if it is a member of the least_int_method_set []; otherwise, it comes out of hindering state after getting the least_int_method_set []. At this point, we conclude that this method is not going to be included in the minimum set. It should be noted that the hindering time of a method is bare minimum.

On receiving the transient reinstatement_point invitation along with the *least_int_method_set []*, an M_S_St, say M_S_St$_j$, registers the following actions. It sets the timer *timer_transient;* sends the transient reinstatement_point invitation to P_i only if P_i belongs to the *least_int_method_set []* and P_i is running in its cubicle. On receiving the reinstatement_point invitation, P_i registers its transient reinstatement_point and informs M_S_St$_j$. On receiving positive response from P_i, M_S_St$_j$ updates $p\text{-}c\_s\_n_i$, resets $hindering_i$, and sends the buffered application_communications to P_i, if any. Alternatively, If P_i is not in the *least_int_method_set []* and P_i is in the cubicle of M_S_St$_j$, M_S_St$_j$ resets $hindering_i$ and sends the buffered application_communication to P_i, if any. For a disconnected Mob_Node, that is a member of *least_int_method_set []*, the M_S_St that has its disconnected reinstatement_point, converts its disconnected reinstatement_point into the required one.

During hindering period, P_i processes m, received from P_j , if following conditions are met: (i) (! buffer$_i$) i.e. P_i has not buffered any application_communication (ii) ($m.psn<=c\_s\_n[j]$) i.e. P_j has not registered its reinstatement_point before sending m (iii) (*cdd_set[]* $_i$[j]=1) P_i is already dependent upon P_j in the current CI or P_j has captured some permanent reinstatement_point after sending *m*. Otherwise, the local M_S_St of P_i buffers *m* for the hindering period of P_i and sets $buffer_i$.

On expiry of timer_transient, if M_S_St$_j$ does not get the positive response to transient reinstatement_point invitation from all pertinent local Mob_Nodes, it informs failure communication to M_S_St$_{in}$and M_S_St$_{in}$issues abort. Alternatively, on expiry of timer_transient, M_S_Stj issues tentative reinstatement_point invitation to the pertinent Mob_Nodes in its cubicle and sets timer_tent.

If some method fails to register its tentative reinstatement_point, then M_S_St$_j$ informs M_S_St$_{in}$ and M_S_St$_{in}$ issues abort. Otherwise, after the timeout of *timer_tent*, M_S_St$_j$ commits the reinstatement_points of the methods of the least_int_method_set [] which are local to its cubicle. On expiry of timer_tent, if M_S_St$_i$ does not get abort massage from M_S_St$_{in}$, it is presumed that all pertinent methods have captured their tentative reinstatement_points successfully; and the etiquette should enter the third phase in which all pertinent methods convert their tentative reinstatement_points into the permanent ones.

We explain the recommended least_int_method CGS_assortment etiquette with the help of an example. In Figure 1, at time t_0, P_5 initiates CGS_assortment procedure and sends invitation to all methods for their causal_depend_arrays[]. At time t_1, P_5 receives the causal_depend_arrays[] from all methods and computes the *least_int_method_set[]* which is {P_4, P_5, P_6}. For the sake of simplicity, the control communications by which the methods send their causal_depend_arrays[] to the originator method P_5 are not shown in the Figure 1. P_5 sends *least_int_method_set []*to all methods and registers its own transient reinstatement_point C_{51}. On receiving *least_int_method_set[]* , *a* method records its transient reinstatement_point if it is a member of *least_int_method_set[]*. When P_4 and P_6 get the *least_int_method_set []*, they find themselves to be the members of the *least_int_method_set []*; therefore, they register their transient reinstatement_points, C_{41} and C_{61}, respectively. When P_1, P_2 and P_3 get the *least_int_method_set []*, they find that they do not have its place in *least_int_method_set []*, therefore, they do not register their transient reinstatement_points. It should be noted that these methods have not sent any application_communication to any method of the least_int_method_set []. In other words, P_5 is not transitively dependent upon them. Therefore, for the sake of consistency, it is not necessary for them to register their reinstatement_points in the current initiation.

A method comes into the hindering state immediately after sending the *cdd_set [] []*. A method comes out of the hindering state only after taking its transient reinstatement_point if it is a member of the least_int_method_set []; otherwise, it comes out of hindering state after getting the least_int_method_set[]. We want to say that the hindering time of a method in this etiquette is negligibly small. Moreover, a method is allowed to perform its normal computation, send application_communications and partially receive them during the hindering period. For example, P_5 receives m_4 during its hindering period. As *cdd_set []* $_5$[6]=1 due to m_2, and receive of m_4 will not alter *cdd_set[]* $_5$[]; therefore P_5 methods m_4. P_2 receives m_{15} from P_3 during its hindering period; *cdd_set[]*$_2$[3]=0 and the receiver of m_{15} can alter *cdd_set[]*$_2$; therefore, P_2 buffers m_{15}. Similarly, P_4 buffers m_{16}. P_4 dispenses m_{16} only after taking its transient reinstatement_point C_{41}. P_2 dispenses m15 after getting the *least_int_method_set []*. P_4 dispenses $m_{7,}$ because, at this moment, it not in the hindering state. Similarly, P_4 processes m_8.

On getting the transient reinstatement_point invitation, a method, say P_6, sets the timer *timer_transient*. If P_6 fails to register its transient reinstatement_point, it informs P_5 and P_5 will issue abort. In this way, if any method fails to register its reinstatement_point in harmonization with others in the first phase, then all the methods need to abort their transient reinstatement_points only and not the tentative reinstatement_points as in other etiquettes [2, 3, 4]. In this way, we are able to significantly diminish the forfeiture of CGS_assortment effort in case of a failure during CGS_assortment. On the other hand, on timeout of *timer_transient* and no abort communication from P_5, it is presumed that all pertinent methods have captured their transient reinstatement_points successfully and the etiquette should enter into the second phase. Therefore, P_6 converts its transient reinstatement_point into tentative one and sets the timer *timer_tent.* If P_6 fails to convert its transient reinstatement_point into tentative one, it informs P_5 and P_5 will issue abort. Similarly, if any other method fails to register its transient reinstatement_point, it will inform P_5 and P_5 will act accordingly. Otherwise, on timeout of *timer_tent*, P_6 converts its tentative reinstatement_point into permanent one. on timeout of *timer_tent*and no abort communication from applicable methods, it is presumed that all pertinent methods have captured their tentative reinstatement_points successfully and the etiquette should enter into the second phase. In this way, we commit the reinstatement_points without much harmonization.

2.3 Performance Analysis of the Proposed Protocol

The obstructive time of Koo-Toueg [7] algorithm may be extraordinarily high due to the formation of CGS_assortment tree and obstructive of processes during the whole of the CGS_assortment procedure. It may be quite disagreeable, specifically in Mobile_DS. In Cao-Singhal algorithm [3], obstructive time is abridged ominously as compared to [7]. The obstructive time of the proposed scheme is similar to [3].It should be noted that the proposed protocol is a three-phase protocol. We add two extra phases, one to collect the dependency vectors and another to take the transient snapshots. First phase is added to compute the exact minimum set in the beginning of the protocol to minimize the obstructive time as in [3]. In order to diminish the loss of CGS_assortment effort, when any process fails to take its transient reinstatement_point in harmonization with others; all relevant processes take transient snapshots in the first phase and convert their transient snapshots into tentative ones in the second phase. In this way, by adding extra synchronization message overhead, we are able to deal with the problem of frequent aborts in coordinating CGS_assortment. We try to minimize the loss of checkpointing effort in case of a fault during CGS_assortment. We want to emphasize that we do not send extra synchronization messages for different phases of the protocol as mentioned in Section 1 and 2. Therefore, the synchronization message overhead in the proposed scheme is less than [3]. We use local timers in place of synchronization messages. Only in case of a fault, synchronization messages are sent in order to abort the algorithm

3. AVERAGE HINDERING TIME AND AVERAGE NUMBER OF APPLICATIONS_COMMUNICATIONS BUFFERED

Suppose, the two M_S_Sts are connected using a 1 Mbps communication link. Each Mob_Node or M_S_St has one method running on it. The length of each frame of reference application_communication is 100 bytes. The average delay on static network for sending system communication is (8*100*1000) / (1000000) = 0.8ms. The hindering time is 2*0.8=1.6 ms. In the suggested etiquette, selective incoming application_communications at a method are blocked during its hindering period. We consider the worst case in which all incoming application_communications are blocked. Blocking period in the suggested scheme is negligibly small; therefore the number of application_communications blocked in the etiquettes is insignificant [Refer Table 1]. It should be noted that the number of application_communication blocked during CGS_assortment depends upon the application_communication sending rate and the capacity of the static communication link. Referring Table 1, we can say that the no. of application_communications buffered during CGS_assortment in the suggested etiquette is negligibly small.

Table 1: Average number of communications buffered during CGS_assortment

Message Sending Rate per second	0.001	0.01	0.1	1	10
Average No. of Messages blocked in the suggested Scheme	$1.6*10^{-6}$	$1.6*10^{-5}$	$1.6*10^{-4}$	$1.6*10^{-3}$	$1.6*10^{-2}$

4. CONCLUSION

We have designed a minimum-process synchronous checkpointing algorithm for mobile distributed system. We try to minimize the intrusion of processes during checkpointing. The obstructive time of a process is bare minimum. During obstructive period, processes can do their normal computations, send messages and can process selective messages. 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 snapshot in coordination with others. We also try to minimize the synchronization messages during checkpointing. In the proposed scheme, no synchronization messages are sent in order to enter the second or third phase of the algorithm.

REFERENCES

1. S. K.M. Chandy and L. Lamport. **“Distributed Snapshots: Determining Global States of Distributed Systems”** ACM Transactions Computer systems vol. 3, no.1. pp.63-75, Feb.1985.
2. 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.

3. Guohong Cao and Mukesh Singhal, **“Mutable Checkpoints: A New Checkpointing Approach for Mobile Computing Systems”**, IEEE Transaction On Parallel and Distributed Systems, vol. 12, no. 2, pp. 157-171, February2001.
4. Guohong Cao and Mukesh Singhal, **“On Coordinated Checkpointing in Distributed Systems”** IEEE Transaction on Parallel and Distributed Systems, vol. 9, no. 12, pp. 1213-1224, December 1998.
5. Weigang Ni, Susan V. Vrbsky and Sibabrata Ray **“Pitfalls in Distributed Non-blocking Checkpointing”**, University of Alabama.
6. Prakash R. and Singhal M. **“Maximal Global Snapshot with concurrent initiators”,** Proc. Sixth IEEE Symp. Parallel and Distributed Processing, pp.344-351, Oct.1994.
7. Koo. R. and S. Toueg. **“Checkpointing and Rollback-Recovery for Distributed Systems”.** IEEE Transactions on Software Engineering, SE-13(1):23-31, January1987.
8. Bidyut Gupta, S. Rahimi and Z. Lui. **“A New High Performance Checkpointing Approach for Mobile Computing Systems”**. IJCSNS International Journal of Computer Science and Network Security, Vol.6 No.5B, May 2006.
9. 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.
10. Ch. D.V. Subba Rao and M. M. Naidu. **“A New, Efficient Coordinated Checkpointing Protocol Combined with Selective Sender-Based Message Logging”**.
11. Nuno Neves and W. Kent Fuchs. **“Adaptive Recovery for Mobile Environments”**, in Proc. IEEE High-Assurance Systems Engineering Workshop, October 21-22, 1996, pp.134-141.
12. Y. Manable. **“A Distributed Consistent Global Checkpoint Algorithm With minimum number of Checkpoints”**. Technical Report of IEICE, COMP97-6(April1997).
13. J. L. Kim and T. Park. **“An efficient protocol for checkpointing recovery in Distributed Systems”** IEEE Transaction on Parallel and Distributed Systems,4(8): pp.955-960, Aug 1993.
14. Elnozahy E.N., Alvisi L., Wang Y.M. and Johnson D.B., **“Survey of Rollback-Recovery Protocols in Message-Passing Systems”,** *ACM Computing Surveys*, vol. 34, no. 3, pp. 375-408, 2002.
15. S. Venkatesan and T.T.-Y. Juang, **“Low Overhead Optimistic Crash Recovery”**, Preliminary version appears in Proc. 11th Int’l Conf. Distributed Computing Systems as “Crash Recovery with Little Overhead,” pp.454- 461,1991.
16. 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.
17. J.L. Kim, T. Park, **“An efficient Protocol for checkpointing Recovery in Distributed Systems”,** IEEE Trans. Parallel and Distributed Systems, pp.955-960, Aug.1993.
18. Mansouri, H., Pathan, A-S.K.: Review of checkpointing and rollback recovery protocols for mobile distributed computing systems. In: Ghosh, U., Rawat D.B., Datta, R., Pathan, A-S. K (eds.) Internet of Things and Secure Smart Environments: Successes and Pitfalls, CRC Press, Taylor& Francis Group (2020).
19. Mansouri, H., Pathan, A.-S.K.: Checkpointing distributed application running on mobile Ad Hoc networks. Int. J. High Perform. Comput. Networking **11**(2), 95–107 (2018).
20. Mansouri, H., Pathan, A.-S.: A resilient hierarchical checkpointing algorithm for distributed systems running on cluster federation. In: Thampi, S.M., Martinez Perez, G., Ko, R., Rawat, D.B. (eds.) SSCC 2019. CCIS, vol. 1208, pp. 99–110. Springer, Singapore (2020).