Journal of
Data Acquisition and Processing

ISSN: 1004-9037
https://sjcjycl.cn/
DOI: 10.5281/zenodo.777969

HANDLING INTERMITTENT TERMINATIONS IN INFALLIBLE RECUPERATION LINE ACCRETION ETIQUETTES FOR FAULT-TOLERANT MOBILE DISTRIBUTED SYSTEMS

Ambreena Muneer[1], Dr S.P.Singh[2]
[1]Research Scholar, School of Data Science and Computer Engineering, Nims University Jaipur, Rajasthan, India, ambreenmuneer3@gmail.com
[2] Professor, School of Data Science and Computer Engineering, Nims University Jaipur, Rajasthan, India, drspsingh2511@gmail.com

Abstract
While dealing with mobile distributed frameworks, we come across some concerns like: mobility, low bandwidth of wireless channels and lack of stable storage on mobile nodes, disconnections, limited battery power and high failure rate of mobile nodes. These concerns make traditional Dependable Recovery Line Compilation (IRL-accretion) techniques designed for Distributed frameworks unsuitable for Mobile environments. In this paper, we design a bottommost implementation algorithm for Mobile Distributed frameworks, where no inoperable reclamation-pinpoints are stockpiled and an effort has been made to optimize the filibustering of implementations. We propose to delay the processing of selective epistles at the receiver end only during the IRL-accretion period. A Process is allowed to perform its normal reckonings and send epistles during its filibustering period. In this way, we try to keep filibustering of implementations to bare bottommost. In order to keep the filibustering time bottommost, we collect the dependency vectors and compute the exact bottommost set in the beginning of the algorithm. The number of implementations that take reclamation-pinpoints is curtaild to 1) avoid awakening of Nm_Nds in doze mode of implementation, 2) curtail thrashing of Nm_Nds with IRL-accretion activity, 3) save limited battery life of Nm_Nds and low bandwidth of wireless channels. In coordinated IRL-accretion, if a single implementation misses to take its reclamation-pinpoint; all the IRL-accretion effort goes waste, because, each implementation has to abort its partially-committed reclamation-pinpoint. In order to take its partially-committed reclamation-pinpoint, an Nm_Nd needs to transfer large reclamation-pinpoint data to its local Nm_SS over wireless channels. The IRL-accretion effort may be exceedingly high due to frequent terminations especially in mobile frameworks. We try to curtail the forfeiture of IRL-accretion effort when any implementation misses to take its reclamation-pinpoint in coordination with others

Keywords: Fault tolerance, consistent global state, coordinated IRL-accretion and mobile frameworks .

1. INTRODUCTION

A distributed framework is one that runs on a collection of machines that do not have shared memory, yet looks to its users like a single computer. The term Distributed frameworks is used to describe a framework with the following characteristics: i) it consists of several computers that do not share memory or a clock, ii) the computers communicate with each other by exchanging epistles over a communication network, iii) each computer has its own memory and runs its own operating framework. A distributed framework consists of a finite set of implementations and a finite set of channels.

In the mobile distributed framework, some of the implementations are running on mobile hosts (Nm_Nds). A Nm_Nd communicates with other nodes of the framework via a special node called mobile support station (Nm_SS) [1]. A cell is a geographical area around a Nm_SS in which it can support an Nm_Nd. A Nm_Nd can change its geographical position freely from one cell to another or even to an area covered by no cell. An Nm_SS 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 Nm_SSs. A static node that has no support to Nm_Nd can be considered as a Nm_SS with no Nm_Nd.

Checkpoint is defined as a designated place in a program at which normal implementation is interrupted specifically to preserve the status information necessary to allow resumption of processing at a later time. IRL-accretion is the implementation of saving the status information. By periodically invoking the IRL-accretion implementation, one can save the status of a program at regular intervals. If there is a failure one may restart reckoning from the last reclamation-pinpoints thereby avoiding repeating reckoning from the beginning. The implementation of resuming reckoning by rolling back to a saved state is called rollback recovery. The reclamation-pinpoint-restart is one of the well-known methods to realize reliable distributed frameworks. Each implementation stockpiles a reclamation-pinpoint where the local state information is stored in the stable storage. Rolling back an implementation and again resuming its execution from a prior state involves overhead and delays the overall completion of the implementation, it is needed to make an implementation rollback to a most recent possible state. So it is at the desire of the user for taking many reclamation-pinpoints over the whole life of the execution of the implementation [6, 29, 30, 31].

In a distributed framework, since the implementations in the framework do not share memory, a global state of the framework is defined as a set of local states, one from each implementation. The state of channels corresponding to a global state is the set of epistles sent but not yet acknowledged. A global state is said to be “consistent” if it contains no orphan epistle; i.e., a epistle whose receive event is recorded, but its send event is lost. To recover from a failure, the framework restarts its execution from a previous consistent global state saved on the stable storage during fault-free execution. This saves all the reckoning done up to the last reclamation-pinpointed state and only the reckoning done thereafter needs to be redone. In distributed frameworks , IRL-accretion can be independent, coordinated [6, 11, 13] or quasi-synchronous [2]. Message Logging is also used for fault tolerance in distributed frameworks [22, 29, 30, 31].

In coordinated or synchronous IRL-accretion , implementations take reclamation-pinpoints 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, implementations take partially-committed reclamation-pinpoints and in the second phase, these are made enduring. The main advantage is that only one enduring reclamation-pinpoint and at most one partially-committed reclamation-pinpoint is required to be stored. In the case of a fault, implementations rollback to last reclamation-pinpointed state.

The coordinated IRL-accretion protocols can be classified into two types: filibustering and non-filibustering. In filibustering algorithms, some filibustering of implementations stockpiles place during IRL-accretion [4, 11, 24, 25] In non-filibustering algorithms, no filibustering of implementations is required for IRL-accretion [5, 12, 15, 21]. The coordinated IRL-accretion algorithms can also be classified into following two categories: bottommost-implementation and all implementation algorithms. In all-implementation coordinated IRL-accretion algorithms, every implementation is required to take its reclamation-pinpoint in an initiation [6], [8]. In bottommost-implementation algorithms, bottommost interacting implementations are required to take their reclamation-pinpoints in an initiation [11].

In bottommost-implementation coordinated IRL-accretion algorithms, an implementation Pi stockpiles its reclamation-pinpoint only if it a member of the bottommost set (a subset of interacting implementation). An implementation Pi is in the bottommost set only if the reclamation-pinpoint initiator implementation 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 IRL-accretion interval [CI] and Pk has not stockpiled its enduring reclamation-pinpoint after sending m. The ith CI of an implementation denotes all the reckoning performed between its ith and (i+1)th reclamation-pinpoint, including the ith reclamation-pinpoint but not the (i+1)th reclamation-pinpoint.

In bottommost-implementation IRL-accretion protocols, some inoperable reclamation-pinpoints are stockpiled or filibustering of implementations stockpiles place. In this paper, we propose a bottommost-implementation coordinated IRL-accretion algorithm for non-deterministic mobile distributed frameworks , where no inoperable reclamation-pinpoints are stockpiled. An effort has been made to curtail the filibustering of implementations and the forfeiture of IRL-accretion effort when any implementation misses to take its reclamation-pinpoint in coordination with others.

Rao and Naidu [26] proposed a new coordinated IRL-accretion protocol combined with selective sender-based epistle logging. The protocol is free from the problem of lost epistles. The term ‘selective’ implies that epistles are logged only within a specified interval known as active interval, thereby reducing epistle logging overhead. All implementations take reclamation-pinpoints at the end of their respective active intervals forming a consistent global reclamation-pinpoint. Biswas & Neogy [27] proposed a IRL-accretion and failure recovery algorithm where mobile hosts save reclamation-pinpoints based on mobility and movement patterns. Mobile hosts save reclamation-pinpoints when number of hand-offs exceed a predefined handoff threshold value. Neves & Fuchs [18] designed a time based loosely synchronized coordinated IRL-accretion protocol that removes the overhead of synchronization and piggybacks integer csn (reclamation-pinpoint sequence number). Gao et al [28] developed an index-based algorithm which uses time-coordination for consistently IRL-accretion in mobile computing environments. In time-based IRL-accretion protocols, there is no need to send extra coordination epistles . However, they have to deal with the synchronization of timers. This class of protocols suits to the applications where implementations have high epistle sending rate.

2. THE PROPOSED IRL-ACCRETION SCHEME

2.1 Basic Idea

The proposed scheme is based on keeping track of direct dependencies of implementations. Similar to [4], initiator implementation collects the direct dependency vectors of all implementations, computes bottommost set, and sends the reclamation-pinpoint request along with the bottommost set to all implementations. In this way, filibustering time has been significantly reduced as compared to [11].

During the period, when an implementation sends its dependency set to the initiator and receives the bottommost set, may receive some epistles , which may add new members to the already computed bottommost set [25]. In order to keep the computed bottommost set intact, We have classified the epistles , acknowledged during the filibustering period, into two types: (i) epistles that alter the dependency set of the receiver implementation (ii) epistles that do not alter the dependency set of the receiver implementation. The epistles in point (i) need to be delayed at the receiver side [25]. The epistles in point (ii) can be processed normally. All implementations can perform their normal reckonings and send epistles during their filibustering period. When an implementation buffers a epistle of former type, it does not implementation any epistle till it receives the bottommost set so as to keep the proper sequence of epistles acknowledged. When an implementation gets the bottommost set, it stockpiles the reclamation-pinpoint, if it is in the bottommost set. After this, it receives the buffered epistles , if any. The proposed bottommost-implementation filibustering algorithm forces zero inoperable reclamation-pinpoints at the cost of very small filibustering.

In bottommost-implementation synchronous IRL-accretion , the initiator implementation asks all communicating implementations to take partially-committed reclamation-pinpoints. In this scheme, if a single implementation misses to take its reclamation-pinpoint; all the IRL-accretion effort goes waste, because, each implementation has to abort its partially-committed reclamation-pinpoint. In order to take the partially-committed reclamation-pinpoint, an Nm_Nd needs to transfer large reclamation-pinpoint data to its local Nm_SS over wireless channels. Due to frequent terminations, total IRL-accretion effort may be exceedingly high, which may be undesirable in mobile frameworks due to scarce resources. Frequent terminations may happen in mobile frameworks due to exhausted battery, abrupt disconnection, or bad wireless connectivity. Therefore, we propose that in the first phase, all concerned Nm_Nds will take mutable reclamation-pinpoint only. Mutable reclamation-pinpoint is stored on the memory of Nm_Nd only. In this case, if some implementation misses to take reclamation-pinpoint in the first phase, then Nm_Nds need to abort their mutable reclamation-pinpoints only. The effort of taking a mutable reclamation-pinpoint is negligible as compared to the partially-committed one. When the initiator comes to know that all relevant implementations have stockpiled their mutable reclamation-pinpoints, it asks all relevant implementations to come into the second phase, in which, an implementation converts its mutable reclamation-pinpoint into partially-committed one. In this way, by increasing small synchronization epistle overhead, we try to reduce the total IRL-accretion effort.

2.2 The Proposed Bottommost-implementation IRL-accretion Algorithm

The The initiator Nm_SS sends a request to all Nm_SSs to send the dd_set vectors of the implementations in their cells. All dd_set vectors are at Nm_SSs and thus no initial IRL-accretion epistles or responses travels wireless channels. On receiving the dd_set [] request, a Nm_SS records the identity of the initiator implementation (say mss_ida) and initiator Nm_SS, sends back the dd_set [] of the implementations in its cell, and sets g_chkpt. If the initiator Nm_SS receives a request for dd_set [] from some other Nm_SS (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 a Nm_SS receives dd_set requests from two Nm_SSs, then it discards the request of the initiator Nm_SS with lower mss_id. Otherwise, on receiving dd_set vectors of all implementations, the initiator Nm_SS computes min_vect [], sends mutable reclamation-pinpoint request along with the min_vect [] to all Nm_SSs. When an implementation sends its dd_set [] to the initiator Nm_SS, it comes into its filibustering state. An implementation comes out of the filibustering state only after taking its mutable reclamation-pinpoint if it is a member of the bottommost set; otherwise, it comes out of filibustering state after getting the mutable reclamation-pinpoint request.

On receiving the mutable reclamation-pinpoint request along with the min_vect [], a Nm_SS, say Nm_SSj, stockpiles the following actions. It sends the mutable reclamation-pinpoint request to Pi only if Pi belongs to the min_vect [] and Pi is running in its cell. On receiving the reclamation-pinpoint request, Pi stockpiles its mutable reclamation-pinpoint and informs Nm_SSj. On receiving positive response from Pi, Nm_SSj updates p-csni, resets filibusteringi, and sends the buffered epistles to Pi, if any. Alternatively, If Pi is not in the min_vect [] and Pi is in the cell of Nm_SSj, Nm_SSj resets filibusteringi and sends the buffered epistle to Pi, if any. For a disconnected Nm_Nd, that is a member of min_vect [], the Nm_SS that has its disconnected reclamation-pinpoint, converts its disconnected reclamation-pinpoint into the required one.
During filibustering period, Pi implementations m, acknowledged from Pj, if following conditions are met: (i) (!buferi) i.e. Pi has not buffered any epistle (ii) (m.psn <=csn[j]) i.e. Pj has not stockpiled its reclamation-pinpoint before sending m (iii) (dd_seti[j]=1) Pi is already dependent upon Pj in the current CI or Pj has stockpiled some enduring reclamation-pinpoint after sending m.
Otherwise, the local Nm_SS of Pi buffers m for the filibustering period of Pi and sets bufferi.

When a Nm_SS learns that all of its implementations in bottommost set have stockpiled their mutable reclamation-pinpoints or at least one of its implementation has missed to reclamation-pinpoint, it sends the response epistle to the initiator Nm_SS. In this case, if some implementation misses to take mutable reclamation-pinpoint in the first phase, then Nm_Nds need to abort their mutable reclamation-pinpoints only. The effort of taking a mutable reclamation-pinpoint is negligible as compared to the partially-committed one. When the initiator comes to know that all relevant implementations have stockpiled their mutable reclamation-pinpoints, it asks all relevant implementations to come into the second phase, in which, an implementation converts its mutable reclamation-pinpoint into partially-committed one.

Finally, initiator Nm_SS sends commit or abort to all implementations. On receiving abort, an implementation discards its partially-committed reclamation-pinpoint, if any, and undoes the updating of data structures. On receiving commit, implementations, in the min_vect [], convert their partially-committed reclamation-pinpoints into enduring ones. On receiving commit or abort, all implementations update their dd_set vectors and other data structures.

2.3 An Example

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

An implementation comes into the filibustering state immediately after sending the *dd_set[]*. An implementation comes out of the filibustering state only after taking its mutable reclamation-pinpoint if it is a member of the bottommost set; otherwise, it comes out of filibustering state after getting the mutable reclamation-pinpoint request. P_4 receives m_4 during its filibustering period. As $dd\_set_4[5]=1$ due to m_3, and receive of m_4 will not alter $dd\_set_4[]$; therefore P_4 implementations m_4. P_1 receives m_5 from P_2 during its filibustering period; $dd\_set_1[2]=0$ and the receive of m_5 can alter $dd\_set_1[]$; therefore, P_1 buffers m_5. Similarly, P_3 buffers m_6. P_3 implementations m_6 only after taking its mutable reclamation-pinpoint. P_1 implementation m_5 after getting the *min_vect* [].P_2 implementations m_7 because at this movement it not in the filibustering state. Similarly, P_3 implementations m_8. At time t_3, P_4 receives responses to mutable check point requests from all relevant implementations (not shown in the Figure 1) and concerns partially-committed reclamation-pinpoint request to all implementations. A implementation in the bottommost set converts its mutable reclamation-pinpoint into partially-committed one. Finally, at time t_4, P_4 receives responses to partially-committed reclamation-pinpoint requests from all relevant implementations (not shown in the Figure 1) and concerns the commit request.

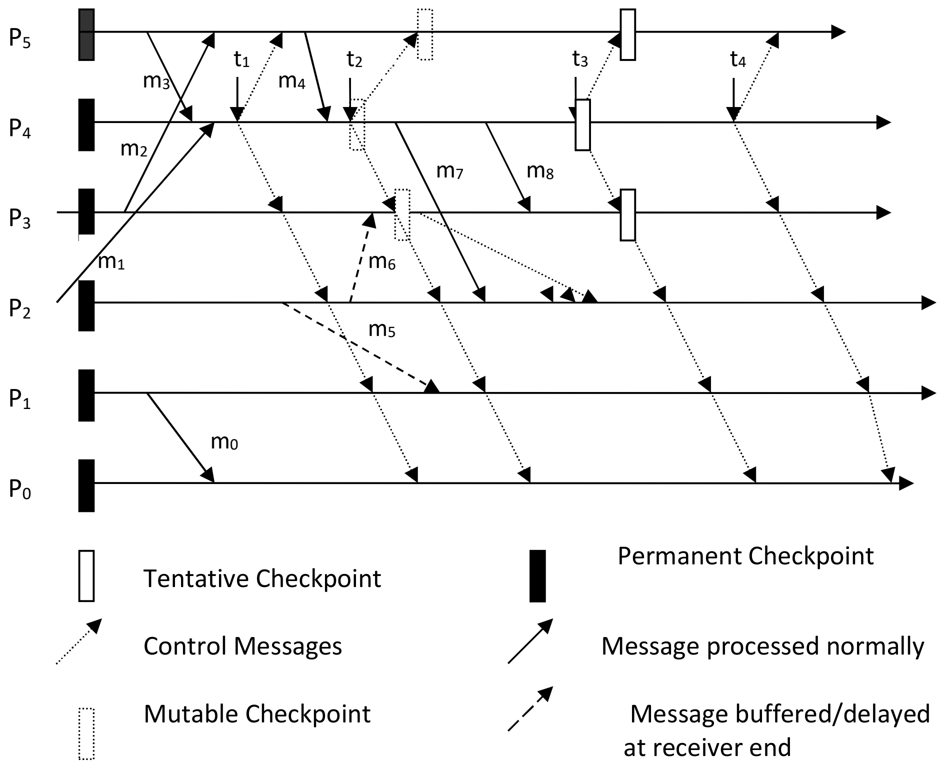

Figure 1 An Example of the proposed Protocol

3. Evaluation of the proposed bottommost-implementation IRL-accretion algorithm

3.1. Computation of average filibustering time and average number of epistles stalled

A The mobile distributed framework considered has N Nm_Nds and M Nm_SSs. Each Nm_SS is a fixed host that has wired and wireless interface. The two Nm_SSs are connected using a 2Mbps communication link. Each Nm_Nd or Nm_SS has one implementation running on it. The length of each framework epistle is 50 bytes. The average delay on static network for sending framework epistle is (8*50*1000)/(2*1000000) = 0.2ms. The filibustering time is 2*0.2=0.4ms. In the proposed algorithm, selective incoming epistles at an implementation are stalled during its filibustering period. We consider the worst case in which all incoming epistles are stalled. Blocking period in the proposed scheme is negligibly small; therefore the number of epistles stalled in the algorithms is insignificant.

Message Sending Rate	0.001	0.01	0.1	1	10
Average No. of Messages stalled in the proposed Scheme	$4*10^{-7}$	$4*10^{-6}$	$4*10^{-5}$	$4*10^{-4}$	$4*10^{-3}$

Table 1:Average number of epistles stalled during IRL-accretion

3.2 Performance of the proposed bottommost-implementation algorithm

We use the following notations for performance analysis of the algorithms:

N_{mss}: number of Nm_SSs.

N_{mh}: number of Nm_Nds.

C_{pp}: cost of sending a epistle from one implementation to another

C_{st}: cost of sending a epistle between any two Nm_SSs.

C_{wl}: cost of sending a epistle from an Nm_Nd to its local Nm_SS (or vice versa).

C_{bst}: cost of broadcasting a epistle over static network.

C_{search}: cost incurred to locate an Nm_Nd and forward a epistle to its current local Nm_SS, from a source Nm_SS.

T_{st}: average epistle delay in static network.

T_{wl}: average epistle delay in the wireless network.

T_{ch}: average delay to save a reclamation-pinpoint on the stable storage. It also includes the time to transfer the reclamation-pinpoint from an Nm_Nd to its local Nm_SS.

N: total number of implementations

N_{min}: number of bottommost implementations required to take reclamation-pinpoints.

N_{mut}: number of inoperable mutable reclamation-pinpoints [5].

N_{ind}: number of inoperable induced reclamation-pinpoints [15].

N_{mutp} number of inoperable mutable reclamation-pinpoints [12]

h: height of the IRL-accretion tree in Koo-Toueg [11] algorithm.

T_{search}: average delay incurred to locate an Nm_Nd and forward a epistle to its current local Nm_SS.

The Blocking Time:

During the time, when a Nm_SS sends the *dd_set* [] vectors and receives the reclamation-pinpoint request, all the implementations in its cell remain in the filibustering period. During the filibustering, an implementation can perform its normal reckonings, send epistles and partially receive them. In the proposed scheme, filibustering period of an implementation is $2T_{st}$.

The Synchronization epistle overhead:

In worst case, it includes the following:

The initiator Nm_SS broadcasts send *dd_set* [], take_mutable_chkpt(), take_partially-committed_chkpt() and commit() epistles to all Nm_SSs: $4C_{bst}$.

The reclamation-pinpoint request epistle from initiator implementation to its local Nm_SS and its response: $2C_{wireless}$.

All Nm_SSs send *dd_set []* of their implementations and response to mutable and partially-committed reclamation-pinpoint request: $3N_{mss}*C_{st}$.

Nm_SSs send reclamation-pinpoint and commit requests to relevant implementations and receive response epistles : $5N_{mh}* C_{wl}$.

Total Message Overhead : $4C_{bst}+ 2C_{wireless} +3N_{mss}*C_{st}+ 5N_{mh}* C_{wl}$.

Number of implementations taking reclamation-pinpoints: In our algorithm, only bottommost number of implementations is required to reclamation-pinpoint.

3.3 Comparison with other algorithms:

	Cao-Singhal [4]	Cao-Singhal [5]	Lalit Kumar et al [15]	Elnozahy et al [8]	P. Kumar et al [12]	Proposed Algorithm
Avg. filibustering Time	$2T_{st}$	0	0	0	0	$2T_{st}$
Average No. of reclamation-pinpoints	N_{min}	$N_{min}+ N_{mut}$	$N_{min}+ N_{ind}$	N	$N_{min}+ N_{mutp}$	N_{min}
Average Message Overhead	$3C_{bst}+ 2C_{wireless} +2N_{mss}* C_{st}+ 3N_{mh}* C_{wl}$.	$2*N_{min} * C_{pp} +C_{bst}$	$3C_{bst}+2 C_{wl}+2N_{mss}* C_{st} +3N_{mh} * C_{wl}$	$2*C_{bst} + N *C_{pp}$	$3C_{bst}+2C_{wl} +(2N_{mss}+p)* C_{st} +3N_{mh} * C_{wl}$	$4C_{bst}+ 2C_{wireless} +3N_{mss}*C_{st}+ 5N_{mh}* C_{wl}$.
Piggybacked Information	Nil	Integer	Integer	Integer	Integer	Integer
Concurrent executions	No	Yes	No	No	No	No

The Koo-Toueg [11] algorithm is a bottommost-implementation coordinated IRL-accretion algorithm for distributed frameworks . It requires implementations to be stalled during IRL-accretion . IRL-accretion includes the time to find the bottommost interacting implementations and to save the state of implementations on stable storage, which may be too long.

In Cao-Singhal algorithm [4], filibustering time is reduced significantly as compared to [15].

P. Kumar [25] finds the problem with algorithm [4]. The algorithms proposed in [5, 12, 15] are non-filibustering, but they suffer from inoperable reclamation-pinpoints. In the proposed scheme, the synchronization epistle is on higher side. We add two extra phases, one to collect the dependency vectors and another to take the mutable reclamation-pinpoints. First phase is added to compute the exact bottommost set in the beginning of the protocol to curtail the filibustering time as in [4] & [25]. In order to curtail the forfeiture of IRL-accretion effort when any implementation misses to take its reclamation-pinpoint in coordination with others, all relevant implementations take mutable reclamation-pinpoints in the first phase and convert their mutable reclamation-pinpoints into partially-committed reclamation-pinpoints in the second phase. In this way, by adding extra synchronization epistle overhead, we are able to deal with the problem of frequent terminations in coordinating IRL-accretion.

4. Conclusions

We have proposed a bottommost implementation coordinated IRL-accretion algorithm for mobile distributed framework, where no inoperable reclamation-pinpoints are stockpiled and an effort is made to curtail the filibustering of implementations. We are able to reduce the filibustering time to bare bottommost by computing the exact bottommost set in the beginning. Furthermore, the filibustering of implementations is reduced by allowing the implementations to perform their normal reckonings and send epistles during their filibustering period. The number of implementations that take reclamation-pinpoints is curtaild to avoid awakening of Nm_Nds in doze mode of implementation and thrashing of Nm_Nds with IRL-accretion activity. It also saves limited battery life of Nm_Nds and low bandwidth of wireless channels. We try to reduce the forfeiture of IRL-accretion effort when any implementation misses to take its reclamation-pinpoint in coordination with others.

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.

[26] Rao, S., & Naidu, M.M, *"A New, Efficient Coordinated Checkpointing Protocol Combined with Selective Sender-Based Message Logging", IEEE/ACS International Conference on Computer Systems and Applications, 2008.*

[27] Biswas S, & Neogy S,"A Mobility-Based Checkpointing Protocol for Mobile Computing System", *International Journal of Computer Science & Information Technology, Vol.2, No.1,pp135-15,2010.*

[28] Gao Y., Deng C., & Che, Y.," An Adaptive Index-Based Algorithm Using Time-Coordination in Mobile Computing", *International Symposiums on Information Processing, pp.578-585*, 2008.

[29] 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.

[30] Praveen Choudhary, Parveen Kumar," Minimum-Process Global-Snapshot Accumulation Etiquette for Mobile Distributed Systems ", International Journal of Advanced Research in Engineering and Technology" Vol. 11, Issue 8, Aug 20, pp.937-948

[31] Deepak Chandra Uprety, Parveen Kumar, Arun Kumar Chouhary,"Transient Snapshot based Minimum-process Synchronized Checkpointing Etiquette for Mobile Distributed Systems",International Journal of Emerging Trends in Engineering Research", Vol 10, No 4, Aug. 2021.