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/ijatcse391042021.pdf
https://doi.org/10.30534/ijatcse/2021/391042021

Treatment of Repeated Letdowns in Coordinated Consistent Recovery Line Compilation for Mobile Distributed Systems

Devarapalli Raghu [1], Parveen Kumar[2]
[1]Research Scholar , Nims University Rajasthan,Jaipur ,India, raghuau@gmail.com
[2] Professor, Department of Computer Science & Information Technology
Nims University Rajasthan Jaipur ,India, parveen.kumar@nimsuniversity.org

ABSTRACT

We put forward a least_int_method (least interactive method) orchestrated CRL-compilation (consistent recovery line compilation) etiquette for non-deterministic Mob_DS (Mobile Distributed Systems); where no inoperable reinstatement-points are recorded. Recurrent terminations of CRL-compilation procedure may happen in Mobile_DS due to exhausted battery, non-voluntary disconnections of Mob_Nodes, or poor wireless connectivity. Therefore, we put forward that in the first stage, all pertinent Mob_Nodes will capture transient reinstatement-point only. Transient reinstatement-point is stored on the memory of Mob_Node only. In this case, if some method fails to capture its reinstatement-point in the first stage, then Mob_Nodes need to abandon their transient reinstatement-points only. In this way, we try to abate the loss of CRL-compilation effort when any method fails to capture its reinstatement-point in harmonization with others. We also try to reduce the CRL-compilation time and intrusion time of methods by limiting CRL-compilation tree which may be formed in other etiquettes [2, 9, 10]. We captured the transitive dependencies during the normal execution by piggybacking causal-dependency-vectors onto computation communications.

Key words : Fault Tolerance, Mobile Computing Systems, Coordinated checkpointing, Rollback Recovery.

1. INTRODUCTION

A Dist-Syst (Distributed System) is an assortment of self-regulating entities that cooperate to solve a problem that cannot be discretely elucidated. A Mob_DS is a Dist-Syst where some of methods are running on mobile nodes (Mob_Nodes), whose location in the network changes with time. The number of methods that capture reinstatement-points is abated to 1) avoid awakening of Mob_Nodes in doze mode of operation, 2) abate thrashing of Mob_Nodes with CRL-compilation activity, 3) save limited battery life of Mob_Nodes and low bandwidth of wireless channels. In least_int_method CRL-compilation etiquettes, some inoperable reinstatement-points are recorded or intrusion of methods records place. In this paper, we put forward a least_int_method orchestrated CRL-compilation etiquette for non-deterministic Mob_DS, where no inoperable reinstatement-points are recorded. An effort has been made to abate the intrusion of methods and harmonization communication overhead. We capture the partial transitive dependencies during the normal execution by piggybacking causal-dependency-vectors onto computation communications. Frequent terminations of CRL-compilation procedure may happen in mobile systems due to exhausted battery, non-voluntary disconnections of Mob_Nodes, or poor wireless connectivity. Therefore, we put forward that in the first stage, all pertinent Mob_Nodes will capture transient reinstatement-point only. Transient reinstatement-point is stored on the memory of Mob_Node only. In this case, if some method fails to capture reinstatement-point in the first stage, then Mob_Nodes need to abandon their transient reinstatement-points only. In this way, we try to abate the loss of CRL-compilation effort when any method fails to capture its reinstatement-point in harmonization with others.

All Communications to and from Mob_Node pass through its local Mob_Supp_St. The Mob_Supp_St maintains the dependency information of the Mob_Nodes which are in its cell. The dependency information is kept in Boolean vector R_i for method P_i. The vector has n bits for n methods. When $R_i[j]$ is set to 1, it represents P_i depends upon P_j. For every P_i, R_i is initialized to 0 except $R_i[i]$, which is initialized to 1. When a method P_i running on an Mob_Node, say Mob_Node_p, obtains a communication from a method P_j, Mob_Node_p's local Mob_Supp_St should set $R_i[j]$ to 1.If P_j has recorded its committed reinstatement-point after forwarding m, $R_i[j]$ is not updated.

Suppose there are methods P_i and P_j running on Mob_Nodes, Mob_Node_i and Mob_Node_j with causal-dependency-vectors R_i and R_j. The causal-dependency-vectors of Mob_Nodes, Mob_Node_i and Mob_Node_j are maintained by their local Mob_Supp_Sts, $Mob_Supp_St_i$ and $Mob_Supp_St_j$. Method P_i running on Mob_Node_i forwards communication m to method P_j running on Mob_Node_j. The communication is first sent to $Mob_Supp_St_i$ (local Mob_Supp_St of Mob_Node_i). $Mob_Supp_St_i$ maintains the causal_dependency_vector R_i of Mob_Node_i. $Mob_Supp_St_i$ appends R_i with communication m and forwards it to $Mob_Supp_St_j$ (local Mob_Supp_St of Mob_Node_j). $Mob_Supp_St_j$ maintains the causal_dependency_vector R_j of Mob_Node_j. $Mob_Supp_St_j$ replaces R_j with bitwise logical OR of causal-dependency-vectors R_i and R_j and forwards m to P_j.

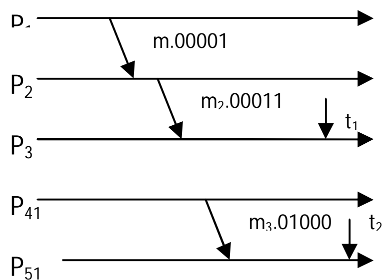

Figure 1. Maintenance of Dependency Vectors

In Figure 1, there are five methods P_1, P_2, P_3, P_4, P_5 with causal-dependency-vectors R_1, R_2, R_3, R_4, R_5 initialized to 00001, 00010, 00100, 01000, and 10000 respectively. Initially, every method depends upon itself. Now method P_1 forwards m to P_2. P_1 appends R_1 with m. P_2 replaces R_2 with the bitwise logical OR of R_1(00001) and R_2(00010), which comes out to be (00011). Now P_2 forwards m2 to P_3 and appends R_2 (00011) with m_2. Before receiving m_2, the value of R_3 at P_3 was 00100. After receiving m_2, P_3 replaces R_3 with the bitwise logical OR of R_2 (00011) and R_3 (00100) and R_3 becomes (00111). Now P_4 forwards m_3 along with R_4 (01000) to P_5. After receiving m_3, R_5 becomes (11000).In this case, if P_3 starts CRL-compilation at t_1, it will compute the partially committed least_int_sett[] (minimum set) equivalent to R_3(00111), which comes out to be {P_1, P_2, P_3}. In this way, partial transitive dependencies are captured during normal computations.

In orchestrated CRL-compilation, if a single method fails to capture its reinstatement-point; all the CRL-compilation effort goes waste, because, each method has to abandon its partially committed reinstatement-point [1, 2]. Furthermore, in order to capture the partially committed reinstatement-point, a Mob_Node needs to transfer large reinstatement-point data to its local Mob_Supp_St over wireless channels. Hence, the loss of CRL-compilation effort may be exceptionally high due to recurrent terminations of CRL-compilation etiquettes especially in mobile systems. In Mob_DS, there remain certain issues like: abrupt disconnection, exhausted battery power, or failure in wireless bandwidth. So there remains a good probability that some Mob_Node may fail to capture its reinstatement-point in harmonization with others. Therefore, we put forward that in the first stage, all methods in the least_int_sett[], capture transient reinstatement-point only. Transient reinstatement-point is stored on the memory of Mob_Node only. If some method fails to capture its reinstatement-point in the first stage, then other Mob_Nodes need to abandon their transient reinstatement-points only. The effort of recording an transient reinstatement-point is insignificant as compared to the partially committed one. In other etiquettes [3, 4], all pertinent methods need to abandon their partially committed reinstatement-points in this situation. Hence the loss of CRL-compilation effort in case of an abandon of the CRL-compilation procedure is dramatically low in the proposed scheme as compared to other orchestrated CRL-compilation schemes for Mob_DS [5, 6].

In this second stage, a method converts its transient reinstatement-point into partially committed one. By using this scheme, we try to abate the loss of CRL-compilation effort in case of abandon of CRL-compilation etiquette in the first stage.

A non-intrusion CRL-compilation etiquette does not require any method to suspend its underlying computation. When methods do not suspend their computation, it is possible for a method to receive a computation communication from another method, which is already running in a new CRL-compilation interval. If this situation is not properly dealt with, it may result in an inconsistency. During the CRL-compilation procedure, a method P_i may receive m from P_j such that P_j has recorded its reinstatement-point for the current instigation whereas P_i has not. Suppose, P_i methods m, and it obtains reinstatement-point request later on, and then it records its reinstatement-point . In that case, m will become orphan in the recorded global state. We put forward that only those communications, which can become orphan, should be buffered at the forwarder's end. When a method records its transient reinstatement-point, it is not allowed to forward any communication till it obtains the partially committed reinstatement-point request. However, in this duration, the method is allowed to perform its normal computations and receive the communications. When a method obtains the partially

committed reinstatement-point request, it is established that every pertinent method has recorded its transient reinstatement-point . Hence, a communication generated for forwarding by a method after getting partially committed reinstatement-point request cannot become orphan. Hence, a method can forward the buffered communications after getting the partially committed reinstatement-point request from the originator_method.

2. THE PROPOSED ETIQUETTE

First stage of the etiquette: proxy

When a method, say P_i, running on a Mob_Node, say Mob_Node_i, initiates a CRL-compilation, it forwards a reinstatement-point instigation request to its local Mob_Supp_St, which will be the alternative Mob_Supp_St. The alternative Mob_Supp_St maintains the causal_dependency_vector of P_i say R_i. On the basis of R_i, the set of dependent methods of P_i is formed, say $S_{least_int_set}$. The alternative Mob_Supp_St broadcasts ckpt ($S_{least_int_set}$) to all Mob_Supp_Sts. When an Mob_Supp_St receive ckpt ($S_{least_int_set}$) communication, it checks, if any methods in $S_{least_int_set}$ are in its cell. If so, the Mob_Supp_St forwards transient reinstatement-point request communication to them. Any method receiving a transient reinstatement-point request records a transient reinstatement-point and forwards a response to its local Mob_Supp_St. After an Mob_Supp_St received all response communications from the methods to which it sent transient reinstatement-point request communications, it forwards a response to the alternative Mob_Supp_St. It should be noted that in the first stage, all methods capture the transient reinstatement-points. For a method running on a static host, transient reinstatement-point is equivalent to partially committed reinstatement-point . But, for an Mob_Node, transient reinstatement-point is different from partially committed reinstatement-point . In order to capture a partially committed reinstatement-point , an Mob_Node has to record its local state and has to transfer it to its local Mob_Supp_St. But, the transient reinstatement-point is stored on the local disk of the Mob_Node. It should be noted that the effort of recording a transient reinstatement-point is very small as compared to the partially committed one. For a disconnected Mob_Node that is a member of least_int_sett[], the Mob_Supp_St that has its disconnected reinstatement-point , considers its disconnected reinstatement-point as the required one.

Second Stage of the Etiquette:

After the substitution Mob_Supp_St has received the response from every Mob_Supp_St, the etiquette enters the second stage. If the alternative Mob_Supp_St learns that all relevant methods have recorded their transient reinstatement-points efficaciously, it asks them to convert their transient reinstatement-points into partially committed ones and also forwards the exact least_int_sett[] along with this request. Alternatively, if originator Mob_Supp_St comes to know that some method has miscarried to capture its reinstatement-point in the first stage, it issues abandon request to all Mob_Supp_St. In this way the Mob_Nodes need to abandon only the transient reinstatement-points, and not the partially committed ones. In this way we try to reduce the loss of CRL-compilation effort in case of abandon of etiquette in first stage.

When an Mob_Supp_St obtains the partially committed reinstatement-point request, it asks all the method in the least_int_sett[], which are also running in itself, to convert their transient reinstatement-points into partially committed ones. When an Mob_Supp_St learns that all relevant method in its cell have recorded their partially committed reinstatement-points successfully, it forwards response to alternative Mob_Supp_St. If any Mob_Node fails to transfer its reinstatement-point data to its local Mob_Supp_St, then the failure response is sent to the alternative Mob_Supp_St; which in turn, issues the abandon communication.

Third Stage of the Etiquette:

Finally, when the alternative Mob_Supp_St learns that all methods in the least_int_sett[] have recorded their partially committed reinstatement-points successfully, it issues commit request to all Mob_Supp_Sts. When a method in the least_int_sett[] gets the commit request, it converts its partially committed reinstatement-point into committed one and discards its earlier committed reinstatement-point , if any.

Massage Handling During CRL-compilation:

When a method records its transient reinstatement-point, it does not forward any massage till it obtains the partially committed reinstatement-point request. This time duration of a method is called its indecision period. Suppose, P_i forwards m to P_j after recording its transient reinstatement-point and P_j has not recorded its transient reinstatement-point at the time of receiving m. In this case, if P_j records its transient reinstatement-point after methoding m, then m will become orphan. Therefore, we do not allow Pi to forward any massage unless and until every method in the least_int_sett[] have recorded its transient reinstatement-point in the first stage. P_i can forward massages when it obtains the partially committed reinstatement-point request; because, at this moment every pertinent method has recorded its transient reinstatement-point and m cannot become orphan. The massages to be sent are buffered at forwarders end. In this duration, a method is allowed to continue its normal computations and receive massages.

3. AN EXAMPLE

The recommended Procedure can be better assumed by the illustration shown in Figure 2. There are six methods (P_0 to P_5) denoted by straight lines. Each method is assumed to have initial committed reinstatement-points with csn equal to “0”. C_{ix} denotes the xth reinstatement-points of P_i. Initial causal-dependency-vectors of P_0, P_1, P_2, P_3, P_4, P_5 are [000001], [000010] [000100], [001000], [010000], and [100000], respectively.

P_0 forwards m_2 to P_1 along with its causal-dependency-vector [000001]. When P_1 obtains m2, it computes its causal-dependency-vector by recording bitwise logical OR of causal-dependency-vectors of P_0 and P_1, which comes out to be [000011]. Similarly, P_2 updates its causal-dependency-vector on receiving m_3 and it comes out to be [000111]. At time t_1, P_2 initiates reinstatement-pointing algorithm with its causal-dependency-vector is [000111]. At time t_1, P_2 finds that it is transitively dependent upon P_0 and P_1. Therefore, P_2 computes the partially-committed minimum set [S_{minset}= {P_0, P_1, P_2}]. P_2 forwards the transient reinstatement-point request to P_1 and P_0 and records its own transient reinstatement-point C_{21}. For an Mob_Node the transient reinstatement-point is stored on the disk of Mob_Node. It should be noted that S_{minset} is only a subset of the minimum set. When P_1 records its transient reinstatement-point C_{11}, it finds that it is dependent upon P_3 due to m_4, but P_3 is not a member of S_{minset}; therefore, P_1 forwards transient reinstatement-point request to P_3. Consequently, P_3 records its transient reinstatement-point C_{31}.

After recording its transient reinstatement-point C_{21}, P_2 generates m_8 for P_3. As P_2 has already capturen its transient reinstatement-point for the current instigation and it has not received the partially-committed reinstatement-point request from the initiator; therefore P_2 buffers m_8 on its local disk. We define this duration as the indecision period of a method during which a method is not allowed to forward any massage. The massages generated for forwarding are buffered at the local disk of the forwarder’s method. P_2 can forwards m_8 only after getting partially-committed reinstatement-point request or abort massages from the initiator method. Similarly, after recording its transient reinstatement-point P_0 buffers m_{10} for its indecision period. It should be noted that P_1 obtains m_{10} only after recording its transient reinstatement-point. Similarly, P_3 obtains m_8 only after recording its transient reinstatement-point C_{31}.A method is allowed to receive all the massages during its indecision period; for example, P_3 obtains m_{11}. A method is also allowed to perform its normal computations during its indecision period.

At time t_2, P_2 obtains responses to transient reinstatement-points requests from all method in the minimum set (not shown in the Figure 3.2) and finds that they have capturen their transient reinstatement-points successfully, therefore, P_2 issues partially-committed reinstatement-point request to all methods. On getting partially-committed reinstatement-point request, methods in the minimum set [P_0, P_1, P_2, P_3] convert their transient reinstatement-points into partially-committed ones and forward the response to initiator method P_2; these method also forward the massages, buffered at their local disks, to the destination methods For example, P_0 forwards m_{10} to P_1 after getting partially-committed reinstatement-point request [not shown in the figure]. Similarly, P_2 forwards m_8 to P_3 after getting partially-committed reinstatement-point request. At time t_3, P_2 obtains responses from the method in minimum set [not shown in the figure] and finds that they have capturen their partially-committed reinstatement-points successfully, therefore, P_2 issues commit request to all method. A method in the minimum set converts its partially-committed reinstatement-point into committed reinstatement-point and discards its old committed reinstatement-point if any.

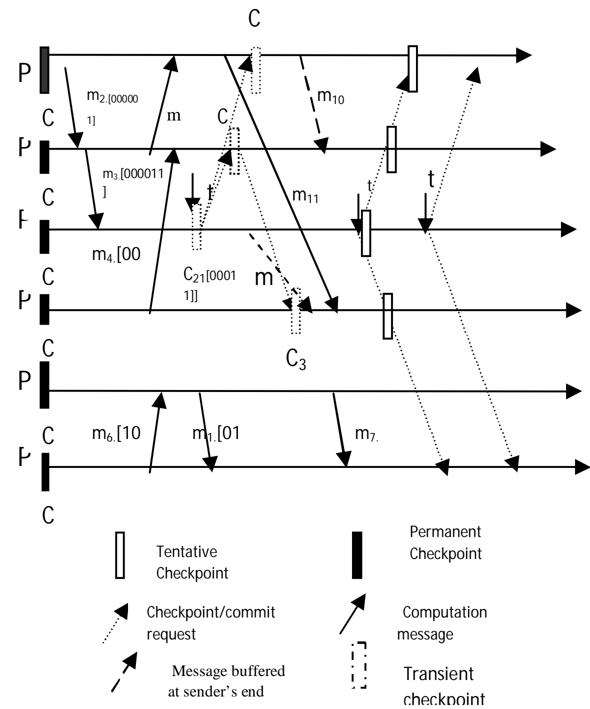

Figure 2

4. CORRECTNESS PROOF

We can show that global state collected by the proposed protocol will be consistent. We can prove the result by contradiction. Suppose there is some orphan message in the recorded global state. We explore

different possibilities with the help of Figure 2 Suppose, P_0 forwards m_{10} after recording its transient reinstatement-point and P_1 obtains m_{10} before recording its transient reinstatement-point. This situation is not possible, because, after recording its transient reinstatement-point P_0 comes into its indecision period and it can not forward any message unless and until it obtains the partially-committed reinstatement-point request. P_2 can issue the partially-committed reinstatement-point request only after getting confirmed that every concerned method (including P_1) has capturen its transient check point. Hence P_1 can not receive m10 before recording its transient reinstatement-point C_{11}. Suppose, P_5 forwards m_{13} to P_3 after C_{50} and P_3 gets m_{13} before C_{31} (not show in the Figure 2). In this case, when P_3 records its transient reinstatement-point C_{31}, it will find that P_5 dose not belong to S_{minset} and P_3 is dependent upon P_5; therefore, P_3 will forward transient reinstatement-point request to P_5 and forward (m_{13}) will also be included in the global state.

5.CONCLUSION

In this paper, we have put forwarded a minimum process consistent recovery line compilation etiquette for non-deterministic Mob_DS, where no inoperable reinstatement-points are recorded and an effort has been made to abate the intrusion of methods. We try to reduce the consistent recovery line compilation time and intrusion time of methods by limiting snapshot compilation tree which may be formed in other etiquettes [2, 9, 10]. We captured the transitive dependencies during the normal execution by piggybacking causal-dependency-vectors onto computation communications. The Z-dependencies are well taken care of in this etiquette. We also try to reduce the loss of CRL-compilation effort when any method fails to capture its reinstatement-point in harmonization with others.

REFERENCES

1. Chandy K.M. and Lamport L., "Distributed snapshots : Determining Global State of Distributed Systems, " ACM Transaction on Computing Systems, vol., 3 No. 1, pp 63-75, February, 1985
2. Koo R. and Tueg S., "Checkpointing and Rollback recovery for Distributed Systems", IEEE Trans. On Software Engineering, Vol. 13 no. 1, pp 23-31, January 1987.
3. Elonzahy E.N., Alvisi L., Wang Y.M. and Johnson D.B., "A survey of Rollback-Recovery protocols in Message-Passing Systems", ACM Computing surveys, vol. 34 no. 3, pp 375-408, 2002.
4. L. Alvisi," Understanding the Message Logging Paradigm for Masking Process Crashes," Ph.D. Thesis, Cornell Univ., Dept. of Computer Science, Jan. 1996. Available as Technical Report TR-96-1577.

5. Lalit Kumar P. Kumar "A synchronous ckeckpointing protocol for mobile distributed systems: probabilistic approach" Int Journal of information and computer security 2007.
6. Cao, M.Singhal, "Mutable Checkpoints: A New Checkpointing Approach for Mobile Computing Systems", IEEE Transactions on Parallel and Distributed system, vol.12, Issue 2, Feb., 2001, pages: 157-172, ISSN: 1045-9219.
7. 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.
8. . M. Singhal and N. Shivaratri, Advanced Concepts in Operating Systems, New York, McGraw Hill, 1994.
9. Cao G. and Singhal M., "On coordinated checkpointing in Distributed Systems", *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no.12, pp. 1213-1225, Dec 1998.
10. Cao G. and Singhal M., "On the Impossibility of Min-process Non-blocking Checkpointing and an Efficient Checkpointing Algorithm for Mobile Computing Systems," *Proceedings of International Conference on Parallel Processing*, pp. 37-44, August 1998.
11. Kumar, P.," A Low-Cost Hybrid Coordinated Checkpointing Protocol for Mobile Distributed Systems", *Mobile Information Systems* pp 13-32, Vol. 4, No. 1. ,2007.
12. 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.