Low-Cost Stable Message Log Purging Algorithm for SBML

Jinho Ahn

***Abstract*—In [1], we have introduced a lightweight SBML protocol to address the following two problems of the original SBML recovery algorithm; it may no longer be progressing in some transient communication error cases and all the message send operations generated after having received some unstable messages should be delayed until they are known to be stable. However, it may make the full log information of each application message recorded on its sender's buffer as well as on the buffer of its immediate dependent. In this paper, we design a novel stable message log purging algorithm to eliminate useless recovery information from immediate dependents' buffers without resulting in any extra control messages. It only piggybacks a variable on original control messages for logging each application message. Finally, we prove the correctness of our algorithm.**

***Index Terms*—Distributed systems, fault-tolerance, sender-based message logging, consistency, log purging.**

I. Introduction

Despite its beneficial features such as requiring no specialized hardware and lowering highly failure-free overhead of synchronous logging [11, 15] with volatile logging at sender's memory, two problems of the original SBML [2, 6, 8, 14] may occur when some transient transmission errors happen [1]. First, when these errors make some received messages be in partially logged states, but their subsequent messages received, in fully logged states, the original SBML's recovery procedure may not progress any longer in case of their receiver's failure. Second, if temporary communication failures force some messages not to be currently fully logged, all the message send operations generated after having received them should be delayed until their receiver can know that they become fully logged on their senders' volatile memories. To address the problems, we have introduced a lightweight SBML protocol to allow a receiver to piggyback small log information for messages received, but not yet fully logged, on each return message for giving the receive sequence number(rsn) assigned to a message to its sender. However, it may make the full log information of each application message recorded on its sender's buffer as well as on the buffer of its immediate dependent. In this paper, we design a novel stable message log purging algorithm to eliminate useless recovery information from immediate dependents' buffers without resulting in any extra control messages. It only piggybacks a variable on original control messages for logging each application message.

Manuscript received June 3, 2012; revised July 5, 2012.

Jinho Ahn is with the Dept. of Computer Science, Kyonggi University, Suwon Gyeonggi, Korea（e-mail: jhahn@kgu.ac.kr).

II. Preliminaries and Main Concepts

A. Previous SBML Protocols

In [1], we have designed a lightweight consistent recovery algorithm for sender-based message logging in distributed systems. Before describing our message log information purging algorithm, let us explain the previous algorithm where the first will be integrated in this section.

The algorithm addressed the following two problems of the original SBML protocol; its recovery procedure may not progress any longer in case of sequential process failures, and all the message send operations generated after having received some unstable messages should be delayed until they are known to be stable. For better understanding, let us explain when the two problems mentioned above may be incurred using Fig. 1 respectively. In this Fig, four processes p1, p2, p3 and p4 are communicating with each other while executing their corresponding tasks. Process p2 takes its latest checkpoint, Chk_2^i, and then receives message m1 from p1, which currently records the partial log information of m1, pl(m1), on its volatile memory. Thus, p2 increments its rsn variable, RSN_2, by one, assigns it to m1 and then returns the rsn value of m1 to p1. Afterwards, p2 receives m2 and m3 from p3 and p4 in order and informs p3 and p4 of their corresponding rsn values respectively in the same manner. However, the two return messages including the rsn values of m1 and m2 cannot still be delivered to their senders, p1 and p3 because of transient communication errors that may normally occur in the distributed system models assumed in this literature [4]. In contrast, p4 receives the rsn value of m3, fully logs m3 on its volatile memory(fl(m3)) and then sends an acknowledgement about the receipt of m3's rsn to p2. At this point, suppose p2 fails and attempts to recover its pre-failure state. The recovery algorithm of the original SBML has p2 restore its state using its latest checkpoint and then obtain all the fully logged messages from their senders. However, p2 can only get the rsn value of m3 from p4 and so not know which messages have been sent to p2 before m3 after checkpoint Chk_2^i. The original SBML couldn't consider this situation and so progress its execution any longer. In order to perform consistent recovery in this example, m3's rsn must be invalidated and all the three messages, handled as partially logged messages.

Second, suppose the original SBML executes according to the scenario of Fig. 1. In this case, due to several transient communication errors from p2 to p1 and p3, p2 may first be informed of p4's receipt of m3's rsn value without knowing whether m1 and m2 are fully logged on their senders properly. Therefore, all message send operations delayed after having received m1 should not be sent even in case of this situation to ensure system consistency. These deferred send operations can begin executing only after p2 have received all the acknowledgements about the receipt of both m1's and m2's rsns in this Fig. This feature can considerably degrade failure-free performance of the entire system.

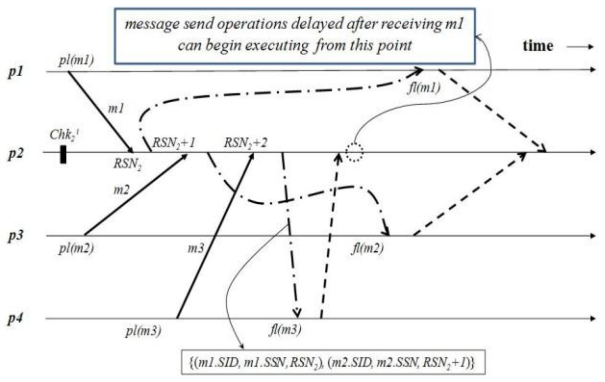

message send operations delayed after receiving m1 can begin executing from this point
time
p1
p2
p3
p4
pl(m1)
m1
fl(m1)
Chk_2^1
RSN_2
RSN_2+1
RSN_2+2
m2
pl(m2)
m3
fl(m2)
pl(m3)
fl(m3)
$\{(m1.SID, m1.SSN, RSN_2), (m2.SID, m2.SSN, RSN_2+1)\}$

Fig. 1. An illustration of execution of the our SBML algorithm

When the return message including the rsn of an application message m received by process p may be lost, there occur two cases our algorithm handles like the original SBML. First, if m's sender q cannot receive the return message within some period of time after having transmitted m, the message m partially logged on q's volatile memory should be retransmitted. Second, if the return message isn't delivered to q after having sent it, p cannot receive any acknowledgement of its receipt from q, retransmitting it to q.

When q receives the return message and then sends the acknowledgement to p, the acknowledgement may be lost. In this case, p re-sends the return message to q, which can give p the acknowledgement without causing any unintended effects.

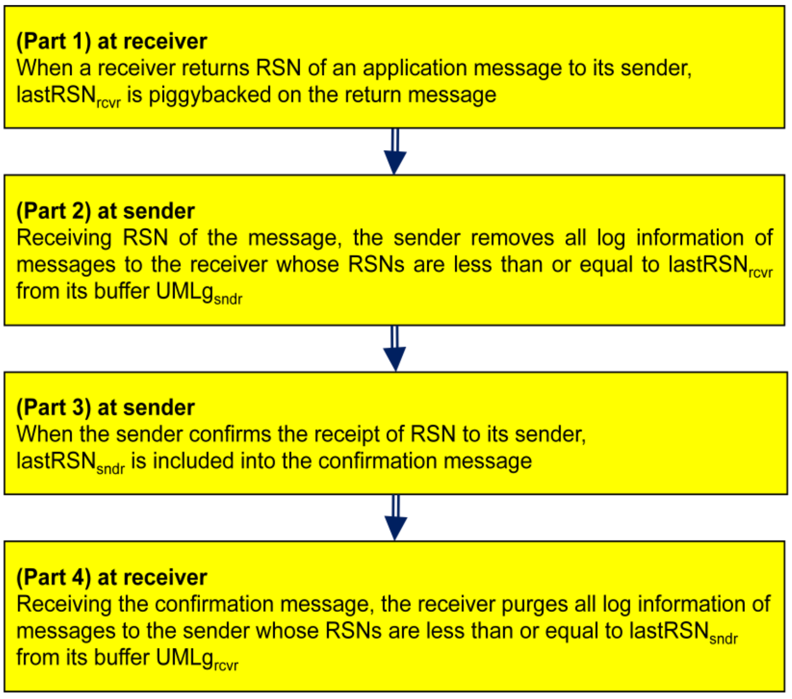
However, our proposed SBML algorithm solves the two problems of the previous SBML by ensuring consistent recovery while handling delayed messages scheduled to be sent much earlier with very low extra overhead even if temporary transmission errors occur. In our algorithm, when p returns the rsn value of the message m to q, it piggybacks on the return message log information for all unstable messages received before m after its latest checkpoint.

In here, unstable message means the message whose receiver cannot currently know whether the rsn of the message is saved on its sender's volatile log properly. On the contrary, a message is called stable that has the opposite property of unstable message.

Also, the log information of each unstable message piggybacked consists of three fields, sender's identifier(SID), send sequence number(SSN) and receive sequence number(RSN) of the message. When receiving the return message, q has to maintain the log information for the unstable messages included in the return message on its volatile memory in addition to updating the rsn value of m into its corresponding log element. As soon as p has received the acknowledgement for m's rsn receipt from q, all the send message operations delayed due to the unstable messages received before m can be performed. For example, as soon as p2 is notified of fully logging m3 on p4's volatile memory in Fig. 1, our algorithm enables all delayed messages scheduled to be sent to be transmitted out because p2 could obtain all the rsn values of the three messages from p4 during recovery unlike the original SBML. If p attempts to take a local checkpoint, it can also allow all the send message operations delayed before this checkpoint to begin executing.

B. Stable Message Log Purging Algorithm

The previous SBML protocol in [1] may save not only the full log information of each application message on its sender's buffer, $Sendlg_{sndr}$, but also on the buffer of another process directly depending on the message, $UMLg_P$. The latter is called immediate dependent. For example, in Fig. 1, after process p2 has received the confirmation messages



(Part 1) at receiver
When a receiver returns RSN of an application message to its sender, $lastRSN_{rcvr}$ is piggybacked on the return message

(Part 2) at sender
Receiving RSN of the message, the sender removes all log information of messages to the receiver whose RSNs are less than or equal to $lastRSN_{rcvr}$ from its buffer $UMLg_{sndr}$

(Part 3) at sender
When the sender confirms the receipt of RSN to its sender, $lastRSN_{sndr}$ is included into the confirmation message

(Part 4) at receiver
Receiving the confirmation message, the receiver purges all log information of messages to the sender whose RSNs are less than or equal to $lastRSN_{sndr}$ from its buffer $UMLg_{rcvr}$

Fig. 2. 4 steps of our message log purging algorithm.

```
Module Msg-Send(data, rcvr) at $P_{sndr}$
    increment $Ssn_{sndr}$ by one ;  assign $Ssn_{sndr}$ to data ;  send m(data, $Ssn_{sndr}$) to $P_{rcvr}$ ;
    $Sendlg_{sndr} \leftarrow Sendlg_{sndr} \cup$ {(rcvr, $Ssn_{sndr}$, -1, data)} ;

Module Msg-Recv(m(ssn, data, sndr)) at $P_{rcvr}$ (including Part 1)
    if($SsnVector_{rcvr}$[m.sndr] < m.ssn) then
        $Rsn_{rcvr} \leftarrow Rsn_{rcvr} + 1$ ;   $SsnVector_{rcvr}$[m.sndr] $\leftarrow$ m.ssn ;
        for all e $\in RSNVector_{rcvr}$ st (e.rsn > $stableRSN_{rcvr}$) do
            UnstableMsgs $\leftarrow$ UnstableMsgs $\cup$ {(e.sid, e.ssn, rcvr, e.rsn)} ;
        send return(m.ssn, $Rsn_{rcvr}$, $lastRSN_{rcvr}$) with UnstableMsgs to $P_{m.sndr}$ ;
        $RSNVector_{rcvr} \leftarrow RSNVector_{rcvr} \cup$ {(m.sndr, m.ssn, $Rsn_{rcvr}$)} ;
        delay all the send message operations generated after having received m ;
        deliver m.data to its corresponding application ;
    else
        find $\exists$e $\in RSNVector_{rcvr}$ st ((i.SID = m.sndr) ^ (i.SSN = m.ssn)) ;
        for all e $\in RSNVector_{rcvr}$ st ((e.rsn < i.RSN) ^ (e.rsn > $stableRSN_{rcvr}$)) do
            UnstableMsgs $\leftarrow$ UnstableMsgs $\cup$ {(e.sid, e.ssn, rcvr, e.rsn)} ;
        send return(m.ssn, i.RSN, $lastRSN_{rcvr}$) with UnstableMsgs to $P_{m.sndr}$ ;

Module RSN-Rcvr(return(ssn, rsn, $lastRSN_{rcvr}$, rcvr, UnstableMsgs)) at $P_{sndr}$ (including Parts 2 and 3)
    find $\exists$e $\in Sendlg_{sndr}$ st ((e.rid = return.rcvr) ^ (e.ssn = return.ssn)) ;   e.rsn $\leftarrow$ return.rsn ;
    $UMLg_{sndr} \leftarrow UMLg_{sndr} \cup$ return.UnstableMsgs ;   send ack(return.rsn)  with $lastRSN_{sndr}$ to $P_{return.rcvr}$ ;
    call Module Remove-LogForstableMsgs(return.rcvr, return.lastRSNrcvr) at itself ;

Module RSN-Ack(ack(rsn, sndr, $lastRSN_{sndr}$)) at $P_{rcvr}$ (including Part 4)
    if($stableRSN_{rcvr}$ < ack.rsn) then
        allow all the send message operations delayed before receiving the message whose rsn value is (ack.rsn+1) to
        begin executing ;
        $stableRSN_{rcvr} \leftarrow$ ack.rsn ;   call Module Remove-LogForstableMsgs(ack.sndr, ack.$lastRSN_{sndr}$) at itself ;

Module Checkpointing() at P
    take its local checkpoint with ($Rsn_P$, $Ssn_P$, $SsnVector_P$, $Sendlg_P$, $UMLg_P$) on the stable storage ;
    allow all the send message operations delayed before this checkpoint to begin executing ;
    $stableRSN_P \leftarrow Rsn_P$ ;   make $RSNVector_P$ an empty set ;

Module Remove-LogForstableMsgs(pid, $lastRSN_{pid}$) at P
    for all e $\in UMLg_P$ st ((e.rid = pid) ^ (e.rsn $\le lastRSN_{pid}$)) do
        $UMLg_P \leftarrow UMLg_P$ – {e} ;
```

Fig. 3. Algorithmic description of our SBML protocol including the proposed message purging algorithm.

The previous SBML protocol in [1] may save not only the full log information of each application message on its sender's buffer, $Sendlg_{sndr}$, but also on the buffer of another process directly depending on the message, $UMLg_P$. The latter is called immediate dependent. For example, in Fig. 1, after process p2 has received the confirmation messages about the receipt of the RSNs of m1 and m2 from p1 and p3, the log information of each message exists on both buffers of its sender and immediate dependent. Thus, it requires an effective log purging algorithm to eliminate useless recovery information from immediate dependents' buffers, which is the focus of our paper that should be addressed. For this purpose, we design a stable message log purging algorithm not to result in any extra control messages with a variable $lastRSN_p$ like in Fig. 2. In here, $lastRSN_p$ is the RSN of the last message which process p has received and seen that was recorded on its sender's buffer. For example, if process p4 sends p2 a new message m4, p2 increments its RSN and return it with $lastRSN_2$ to p4, which can remove the log information of m1 and m2 from its buffer.

Also, if there exist some messages previously received by p4 and p2 is the immediate dependent of them, p2 can eliminate the redundant log information of the messages from its buffer $UMLg_2$ after p4 sends p2 the acknowledgement of m4's RSN with $lastRSN_4$. From this example, we can see that our proposed algorithm doesn't need any extra interaction among processes.

The formal algorithmic description of our log purging algorithm is integrated into the previous protocol of [1] in Fig. 3. Parts 1 to 4 in Fig. 2 are highlighted with underlines in Fig. 3.

C. Correctness Proof

Theorem 1. Even after the proposed message log purging algorithm has performed in our SBML protocol, no useful log information for recovering future failures will be eliminated.

Proof: Let us prove this theorem by contradiction. Suppose that the proposed algorithm may purge the log information useful for future recoveries. As mentioned in section IV, the algorithm forces each process *p* to eliminate log information from its volatile memory only in the following two cases.

Case 1: p receives the return message with the RSN of an application message m p has sent to another process q.

In this case, lastRSN_q was piggybacked on the return message. Thus, p removes from UMLg_p all lge(o)s such that lge(o).rid is q and lge(o).rsn is less than or equal to lastRSN_q. In here, lge(o) consists of four fields, (sid, ssn, rid, rsn), of message o. As every field value of lge(o) is also recorded on $\text{Sendlg}_{\text{lge(o).sid}}$, all these lge(o)s need no longer be kept in UMLg_p in case of failure of process q. Thus, lge(o) isn't useful for its future recoveries.

Case 2: p receives the confirmation message about the receipt of the RSN of an application message m p has received from another process q.

In this case, lastRSN_q was piggybacked on the confirmation message. It is similar to case 1.

Therefore, the proposed algorithm removes only redundant log information on buffers of immediate dependents in any case. This contradicts the hypothesis.

III. Conclusion

This paper presented a redundant stable message log purging algorithm only to piggyback a variable on the return message and confirmation message of the RSN of each application message without requiring any extra message interaction. Also, we proved that our algorithm never violates consistency condition in any case. Therefore, we believe that this algorithm can considerably improve availability of the buffer of each process with little overhead if the SBML protocol in [1] is combined with the proposed one.

References

[1] J. Ahn, "Lightweight Consistent Recovery Algorithm for Sender-based Message Logging in Distributed Systems," *IEICE Transactions on Information and Systems*. 2011, vol. E94-D, no. 8, pp. 1712-1715.

[2] A. Bouteiller, F. Cappello, T. Herault, G. Krawezik, P. Lemarinier, and F. Magniette, "MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging," In: *Proc. of the Int'l Conf. on High Performance Networking and Computing*, 2003.

[3] D. Buntinasd, C. Coti, T. Herault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguez, and F. Cappello, "Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI Protocols," *Future Generation Computer Systems.* 2008, vol. 24, pp.73-84.

[4] K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Transactions on Computer Systems*. 1985, vol. 3, no. 1, pp. 63-75.

[5] E. Elnozahy, L. Alvisi, Y. Wang, and D. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys*. 2002, vol. 34, no. 3, pp. 375-408.

[6] D. Johnson and W. Zwaenpoel, "Sender-based message logging," In: *Proc. of Int'l Symp on Fault-Tolerant Computing*. 1987, pp. 14-19.

[7] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM.* 1978, vol. 21, pp. 558-565.

[8] T. LeBlanc, R. Anand, E. Gabriel, and J. Subhlok, "VolpexMPI: an MPI library for execution of parallel applications on volatile nodes," *Lecture Notes in Computer Science.* 2009, vol. 5759, pp. 124-133.

[9] H. F. Li, Z. Wei, and D. Goswami, "Quasi-atomic recovery for distributed agents," *Parallel Computing.* 2009, vol. 32, pp. 733-758.

[10] Y. Luo and D. Manivannan, "FINE: a fully informed and efficient communication-induced checkpointing protocol for distributed systems," *J. Parallel Distrib. Comput.* 2009, vol. 69, pp. 153-167.

[11] M. Powell and D. Presotto, "Publishing: a reliable broadcast communication mechanism," In: *Proc. Of the 9th International Symposium on Operating System Principles*. 1983, pp. 100-109.

[12] R. D. Schlichting and F. B. Schneider, "Fail-stop processors: an approach to designing fault-tolerant distributed computing systems," *ACM Transactions on Computer Systems.* 1985, vol. 1, pp. 222-238.

[13] R. E. Strom and S. A. Yemeni, "Optimistic recovery in distributed systems," *ACM Transactions on Computer Systems*. 1985, vol. 1, pp. 204-226.

[14] J. Xu, R. B. Netzer, and M. Mackey, "Sender-based message logging for reducing rollback propagation," In: *Proc. of the 7th International Symposium on Parallel and Distributed Processing*. 1995, pp. 602-609.

[15] B. Yao, K. Ssu, and W. Fuchs, "Message logging in mobile computing," In: *Proc. of the 29th International Symposium on Fault-Tolerant Computing*. 1999, pp. 14-19.