IEICE TRANS. INF. & SYST., VOL.E94–D, NO.8 AUGUST 2011

LETTER

Lightweight Consistent Recovery Algorithm for Sender-Based Message Logging in Distributed Systems

Jinho AHN[†a)], ***Member***

SUMMARY Sender-based message logging (SBML) with checkpointing has its well-known beneficial feature, lowering highly failure-free overhead of synchronous logging with volatile logging at sender's memory. This feature encourages it to be applied into many distributed systems as a low-cost transparent rollback recovery technique. However, the original SBML recovery algorithm may no longer be progressing in some transient communication error cases. This paper proposes a consistent recovery algorithm to solve this problem by piggybacking small log information for unstable messages received on each acknowledgement message for returning the receive sequence number assigned to a message by its receiver. Our algorithm also enables all messages scheduled to be sent, but delayed because of some preceding unstable messages to be actually transmitted out much earlier than the existing ones.

key words: *distributed systems, fault-tolerance, message logging, checkpointing, scalability, consistent recovery*

1. Introduction

Sender-based message logging (SBML) with checkpointing is being used as a low-cost transparent rollback-recovery technique in many fields such as mobile computing, cluster and grid computing, sensor network and so on [1], [2], [4]–[7]. This popularity comes from its requiring no specialized hardware and considerably alleviating the normal operation overhead of synchronous logging on stable storage by volatile logging at sender's memory. However, we have identified two problems of the original SBML when some transient transmission errors occur, which can normally be assumed in this literature [3]. First, when these errors make some received messages partially logged, but their subsequently received messages fully logged, 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. All of these features may enormously reduce performance of the entire system. In this paper, we present a lightweight SBML algorithm to solve the two problems. This algorithm enables 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. Our algorithm also enables all messages scheduled to be sent, but delayed by some preceding unstable messages to be actually transmitted out much earlier than the existing ones.

Manuscript received February 10, 2011.
Manuscript revised April 9, 2011.

[†]The author is with the Department of Computer Science, Kyonggi University, Suwon-si Gyeonggi-do, Korea.
a) E-mail: jhahn@kyonggi.ac.kr

DOI: 10.1587/transinf.E94.D.1712

2. System Model

A distributed computation consists of a set P of n (n > 0) sequential processes executed on hosts in the system and there is a distributed stable storage that every process can always access that persists beyond processor failures, thereby supporting recovery from failure of an arbitrary number of processors [3]. Processes have no global memory and global clock. The system is asynchronous: each process is executed at its own speed and communicates with each other only through messages at finite but arbitrary transmission delays. Exchanging messages may temporarily be lost but, eventually delivered in FIFO order. We assume that the communication network is immune to partitioning and hosts fail according to the fail stop model where every crashed process on them halts its computation with losing all contents of its volatile memory [3]. The execution of each process is piecewise deterministic [3]: at any point during the execution, a state interval of the process is determined by a non-deterministic event, which is delivering a received message to the appropriate application. The k-th state interval of process p, denoted by si_p^k(k > 0), is started by the delivery event of the k-th message m of p, denoted by dev_p^k(m). Therefore, given p's initial state, si_p^0, and the non-deterministic events, [dev_p^1, dev_p^2, ..., dev_p^i], its corresponding state s_p^i is uniquely determined. Let p's state, s_p^i = [si_p^0, si_p^1, ..., si_p^i], represent the sequence of all state intervals up to si_p^i. s_p^i and s_q^j (p ≠ q) are mutually consistent if all messages from q that p has delivered to the application in s_p^i were sent to p by q in s_q^j, and vice versa [3]. A set of states, which consists of only one state for every process in the system, is a globally consistent state if any pair of the states is mutually consistent.

3. The Proposed SBML Algorithm

3.1 Problems of the Original SBML

For better understanding, let us explain when the two problems mentioned above may be incurred using Figs. 1 and 2

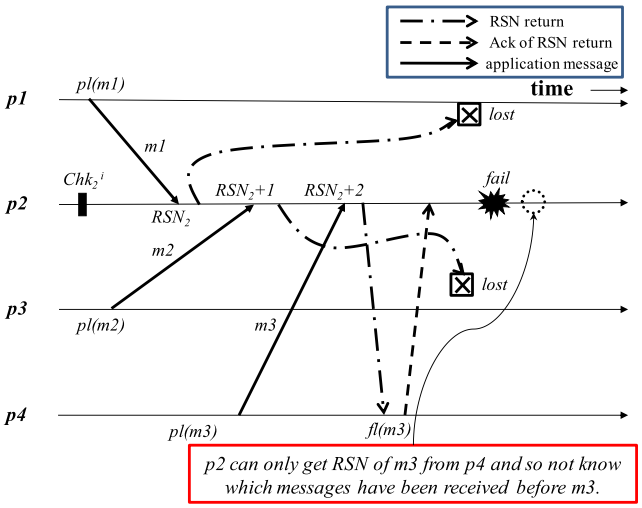

Fig. 1 An example of no progression of the original SBML algorithm in case of transient communication errors.

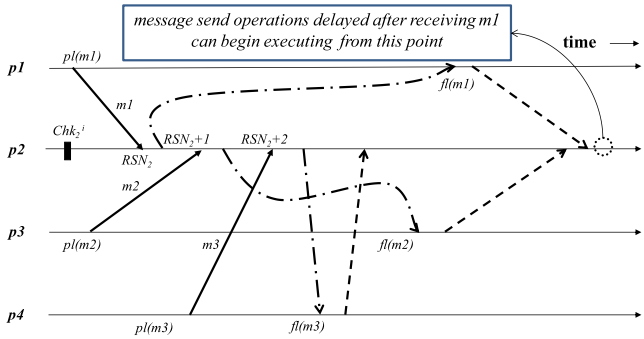

Fig. 2 An example of delayed message send operations incurred in the original SBML algorithm.

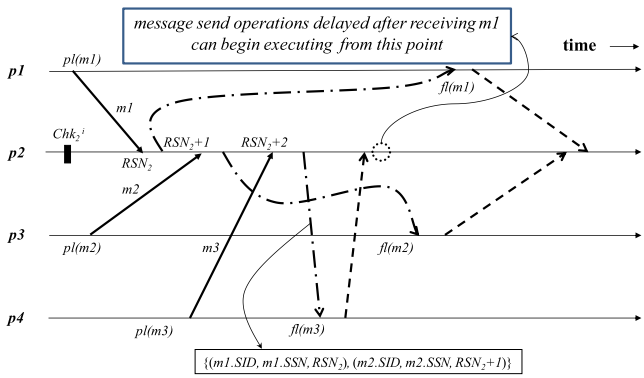

Fig. 3 An example of execution of our SBML algorithm.

respectively. In Fig. 1, 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, like in this figure because of transient communication errors that may normally occur in the distributed system models assumed in this literature [3]. 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. 2. In this case, due to several transient communication errors from p2 to p1 and p3 like in Fig. 1, 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 Fig. 2. This feature can considerably degrade failure-free performance of the entire system.

3.2 Basic Concepts

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 like in **Module** RSN-Rcvr() in Fig. 4.

However, as mentioned in Sect. 3.1, 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

Module MSG-SEND($data$, $rcvr$) AT PROCESS P_{sndr}
  increment Ssn_{sndr} **by** one ; **assign** Ssn_{sndr} **to** $data$;
  send $m(data, Ssn_{sndr})$ **to** P_{rcvr} ;
  $Sendlg_{sndr} \leftarrow Sendlg_{sndr} \cup \{(rcvr, Ssn_{sndr}, \text{-}1, data)\}$;

Module MSG-RECV($m(ssn, data, sndr)$) AT PROCESS P_{rcvr}
  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 $return(m.ssn, Rsn_{rcvr})$ 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 $\exists i \in RSNVector_{rcvr}$ **st** $((i.SID = m.sndr) \wedge (i.SSN = m.ssn))$;
    for all $e \in RSNVector_{rcvr}$ **st** $((e.rsn < i.RSN) \wedge (e.rsn > stableRSN_{rcvr}))$
      do
        $UnstableMsgs \leftarrow UnstableMsgs \cup \{(e.sid, e.ssn, rcvr, e.rsn)\}$;
    send $return(m.ssn, i.RSN)$ with $UnstableMsgs$ **to** $P_{m.sndr}$;

Module RSN-RCVR($return(ssn, rsn, rcvr, UnstableMsgs)$) AT PROCESS P_{sndr}
  find $\exists e \in Sendlg_{sndr}$ **st** $((e.rid = return.rcvr) \wedge (e.ssn = return.ssn))$;
  $e.rsn \leftarrow return.rsn$; $UMLg_{sndr} \leftarrow UMLg_{sndr} \cup return.UnstableMsgs$;
  send $ack(return.rsn)$ **to** $P_{return.rcvr}$;

Module RSN-ACK($ack(rsn)$) AT PROCESS P_{rcvr}
  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 ;
    $stableRSN_{rcvr} \leftarrow ack.rsn$;

Module CHECKPOINTING() AT PROCESS 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 ;

Fig. 4 Message logging procedures.

log information for all unstable messages received before m after its latest checkpoint like in **Module** MSG-RECV() in Fig. 4. 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 four fields, sender's identifier (SID), receiver's identifier (RID), 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 like in **Module** RSN-RCVR() in Fig. 4. 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 like in **Module** RSN-ACK() in Fig. 4. For example, as soon as p2 is notified of fully logging m3 on p4's volatile memory in Fig. 3, 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 in Fig. 1 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. The detailed formal description of our algorithm is shown in Figs. 4 and 5.

Module RECOVERY() AT PROCESS P
  restore its latest checkpointed state with (Rsn_P, Ssn_P, $SsnVector_P$,
    $Sendlg_P$, $UMLg_P$) **from** stable storage ;
  broadcast each a recovery request $request$ **to** every other process ;
  while recovery replies aren't received from all the other processes **do**
    put fully logged messages for P piggybacked
      on each reply r into $flog_P$ in RSN order ;
    put partially logged messages for P piggybacked
      on r into $plog_P$ in FIFO order ;
    put log information for unstable messages to P piggybacked
      on r into $usmsgs_P$ in FIFO order ;
  for all $e \in usmsgs_P$ **do**
    if($\exists i \in plog_P$ **st** $((e.SID = i.SID) \wedge (e.SSN = i.SSN))$) **then**
      $i.RSN \leftarrow e.RSN$; $flog_P \leftarrow flog_P \cup \{i\}$; $plog_P \leftarrow plog_P$ - $\{i\}$;
  for all $e \in flog_P$ **st** ($e.RSN = Rsn_P$) **do**
    increment Rsn_P **by** one ; $SsnVector_P[e.SID] \leftarrow e.SSN$;
    $RSNVector_P \leftarrow RSNVector_P \cup \{(e.SID, e.SSN, Rsn_P)\}$;
    deliver $e.data$ **to** its corresponding application ; $flog_P \leftarrow flog_P$ - $\{e\}$;
  $stableRSN_P \leftarrow Rsn_P$;
  while $plog_P$ is a non-empty set **do**
    randomly select $\exists e$ in $plog_P$ **st** ($e.SSN = SsnVector_P[e.SID]$+1) ;
    call Module MSG-RECV($e.SSN$, $e.data$, $e.SID$) at Process P ;
    $plog_P \leftarrow plog_P$ - $\{e\}$;

Module RREQEST-RCVR($request(rcvr)$) AT PROCESS P_{live}
  put fully and partially logged messages for $request.rcvr$ in $Sendlg_{live}$
    and log information for unstable messages sent to $request.rcvr$
    in $UMLg_{live}$ **into** a reply r ;
  send r **to** $P_{request.rcvr}$;

Fig. 5 Recovery procedures.

Lemma 1. After a process p having received a message m from another process q receives an acknowledgement about the receipt of m's rsn from q in our algorithm, there exists no orphan message in case of p's failure even if p has begun executing all the message send operations delayed to be sent due to any unstable message received before m after p's latest checkpoint.

Proof. Suppose the set of all the unstable messages p received before m after p's latest checkpoint is denoted by $USMSGS_p(m)$. In order to make no orphan message in case of p's failure, all the message send operations depending on any unstable message $\in USMSGS_p(m)$ must be able to be regenerated during recovery after they have been executed during failure-free operation. Therefore, to prove lemma 1, after a process p has received an acknowledgement about the receipt of m's rsn from q, our algorithm must be able to give p the rsn of any message $l \in USMSGS_p(m)$ during p's recovery procedure. The proof proceeds by induction on the number of all the unstable messages in $USMSGS_p(m)$, denoted by $NUM\_OF(USMSGS_p(m))$.

[Base case]

In this case, there is one unstable message l and there are two cases we should consider.

Case 1: l's sender r has received l's rsn and fully logged l on its volatile memory before p's failure.

In this case, p can trivially obtain l's rsn from r.

Case 2: l's sender r hasn't received l's rsn before p's failure.
In this case, as l's rsn was piggybacked on the return message including m's rsn to m's sender q and has been recorded on q's volatile memory before sending an acknowledgement about the return message to p, p can get l's rsn from q.

Therefore, p can obtain the rsn of any message $l \in USMSGS_p(m)$ during p's recovery procedure in all the above cases.

[Induction hypothesis]

We assume that the theorem is true for p in case that $NUM\_OF(USMSGS_p(m)) = k$.

[Induction step]

If the rsn of $(k+1)$-th unstable message can be given to p during p's recovery because p can acquire all rsns of the other k unstable messages by induction hypothesis, the lemma is true for p in case that $NUM\_OF(USMSGS_p(m)) = k+1$. We assume that the $(k+1)$-th unstable message is l. The following case is similar to the base case mentioned above. Therefore, p can obtain the rsn of any message $l \in USMSGS_p(m)$ during p's recovery procedure.

By the induction, there exists no orphan message in case of p's failure even if p has begun executing all the message send operations delayed to be sent due to any unstable message $\in USMSGS_p(m)$. □

Theorem 1. Our algorithm can perform consistent recovery in case of sequential process failures.

Proof. We prove this theorem by contradiction. Assume that consistent recovery may be impossible in case of a single failure of a process p at a time in our algorithm. This assumption means there are one or more message send operations, denoted by $ORPHANOPS_p$, that a process p has performed before its failure, but cannot regenerate like in its failure-free execution even after the recovery procedure of this algorithm executed. Suppose among all the message send operations, operation o is generated as the last in order and l is the most recent message that p has received right before generating the operation o. There are two cases to be considered.

Case 1: l is an unstable message received before p's failure.
In this case, as operation o has been executed during failure-free operation, there was at least one stable message m received by p after l according to our algorithm mentioned in Sect. 3.2. By lemma 1, there exists no orphan message in case of p's failure even if p has begun executing all the message send operations delayed to be sent due to any unstable message $\in USMSGS_p(m)$. Thus, as m's sender has kept the rsn of every unstable message $\in USMSGS_p(m)$ including l on its volatile memory even in case of p's failure, all the operations in $ORPHANOPS_p$ can still be regenerated.

Case 2: l is a stable message received before p's failure.
In this case, as l's sender has received a return message including l's rsn with the rsn of every unstable message $\in USMSGS_p(l)$ from p and recorded all the rsns on its volatile memory, all the operations in $ORPHANOPS_p$ can still be regenerated.

Therefore, consistent recovery is possible in all the cases. This contradicts the hypothesis. □

4. Conclusion

This paper identifies the two drawbacks of the original SBML incurred by transient communication errors, i.e., 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. In order to solve them, we presented a lightweight SBML algorithm to have the following desirable feature. When a process receiving a message m sends each return message for notifying m's sender of its rsn in this algorithm, the rsns of all the unstable messages received before m are included in the return message. This feature can make its consistent recovery procedure ongoing in case of failures. Additionally, the algorithm enables all messages scheduled to be sent, but delayed because of some preceding unstable messages for satisfying the consistency condition to be actually transmitted out much earlier compared with the existing ones.

References

[1] J. Ahn, "Scalable message logging algorithm for geographically distributed broker-based sensor networks," Proc. Int'l Conf. on Computers And Their Applications In Industry and Engineering, pp.279–284, 2010.

[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," Proc. Int'l Conf. on High Performance Networking and Computing, 2003.

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

[4] D. Johnson and W. Zwaenpoel, "Sender-based message logging," Int'l Symp. on Fault-Tolerant Computing, pp.14–19, 1987.

[5] T. LeBlanc, R. Anand, E. Gabriel, and J. Subhlok, "VolpexMPI: An MPI library for execution of parallel applications on volatile nodes," Lect. Notes Comput. Sci., vol.5759, pp.124–133, 2009.

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

[7] B. Yao, K. Ssu, and W. Fuchs, "Message logging in mobile computing," Proc. 29th International Symposium on Fault-Tolerant Computing, pp.14–19, 1999.