Lightweight Log Management Algorithm for Removing Logged Messages of Sender Processes With Little Overhead

JINHO AHN
Department of Computer Science
Kyonggi University
San 94-6 Yiuidong, Yeongtonggu, Suwonsi Gyeonggido 443-760
Republic of Korea
jhahn@kyonggi.ac.kr

Abstract: Sender-based message logging allows each message to be logged in the volatile storage of its corresponding sender. This behavior avoids logging messages on the stable storage synchronously and results in lower failure-free overhead than receiver-based message logging. However, in the first approach, each process should keep in its limited volatile storage the log information of its sent messages for recovering their receivers. In this paper, we propose a 2-step algorithm to efficiently remove logged messages from the volatile storage while ensuring the consistent recovery of the system in case of process failures. As the first step, the algorithm eliminates useless log information in the volatile storage with no extra message and forced checkpoint. But, even if the step has been performed, the more empty buffer space for logging messages in future may be required. In this case, the second step forces the useful log information to become useless by maintaining a vector to record the size of the information for every other process. This behavior incurs fewer additional messages and forced checkpoints than existing algorithms. Experimental results verify that our algorithm significantly performs better than the traditional one with respect to the garbage collection overhead.

KeyWords: Distributed systems, Fault-tolerance, Rollback recovery, Sender-based message logging, Checkpointing, Garbage collection

1 Introduction

As long-running scientific and business applications [4, 6, 8, 9, 10, 17, 23] are executed on large-scale distributed systems composed of hundreds or thousands of independent computers, process failure may become the most critical issue [2, 3, 21]. To address the issue, the systems use log-based rollback recovery as a cost-effective and transparent fault-tolerance technique, in which each process periodically saves it local state by or without synchronizing with other processes, and logs each received message [1, 11, 14, 15, 18]. If a process crashes, the technique creates a new process and allows the process to restore its consistent state and replay its previously received messages beyond the state [13]. Message logging protocols are classified into two approaches, i.e., sender-based and receiver-based message logging, depending on which process each message is logged by [13]. First, receiver-based message logging approach [20, 26] logs the recovery information of every received message to the stable storage before the message is delivered to the receiving process. Thus, the approach simplifies the recovery procedure of failed processes. However, its main drawback is the high failure-free overhead caused by synchronous logging.

Sender-based message logging approach [5, 12, 25] enables each message to be logged in the volatile memory of its corresponding sender for avoiding logging messages to the stable storage synchronously. Therefore, it reduces the failure-free overhead compared with the first approach. But, the second approach forces each process to maintain in its limited volatile storage the log information of its sent messages required for recovering receivers of the messages when they crash. Therefore, the sender-based message logging approach needs an efficient garbage collection algorithm to have the volatile memory of each process for message logging become full as late as possible because, otherwise, the technique forces the message log in the memory to be frequently flushed to stable storage or requires a large number of additional messages and forced checkpoints for removing the log.

Existing sender-based message logging protocols use one between two message log management algorithms to ensure system consistency despite future failures according to each cost like in step 2 of figure 1. The first algorithm just flushes the message log

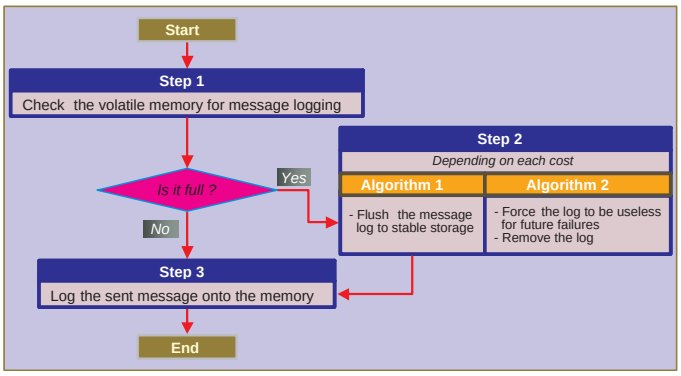

Figure 1: Log management algorithms of previous sender-based message logging protocols

to the stable storage. It is very simple, but may result in a large number of stable storage accesses during failure-free operation and recovery. The second algorithm forces messages in the log to be useless for future failures and then removes them. In other words, the algorithm checks whether receivers of the messages has indeed received the corresponding messages and then taken no checkpoint since. If so, it forces the receivers to take their checkpoints. Thus, this behavior may lead to high communication and checkpointing overheads as inter-process communication rate increases. In this paper, we propose a 2-step algorithm to efficiently remove logged messages from the volatile storage while ensuring the consistent recovery of the system in case of process failures. As the first step, the algorithm eliminates useless log information in the volatile storage with no extra message and forced checkpoint. But, even if the step has been performed, the more empty buffer space for logging messages in future may be required. In this case, the second step forces the useful log information to become useless by maintaining a vector to record the size of the information for every other process. This algorithm can choose a minimum number of processes participating in the garbage collection based on the vector. Thus, this behavior incurs fewer additional messages and forced checkpoints than the existing algorithms.

The remainder of the paper is organized as follows. In sections 2 and 3, we describe the system model and explain our log management algorithm. Sections 4 and 5 prove the correctness of the proposed algorithm and verify its effectiveness by showing results of the experiments respectively. Finally, section 6 concludes this paper.

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 stable storage that every process can always access that persists beyond processor failures [13]. 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. We assume that the communication network is immune to partitioning and hosts fail according to the fail stop model [22]. Events of processes occurring in a failure-free execution are ordered using Lamport's *happened before* relation [16]. The execution of each process is *piecewise deterministic* [24]: 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, $[{dev_p}^1, {dev_p}^2, \cdots, {dev_p}^i]$, its corresponding state ${s_p}^i$ is uniquely determined. Let p's state, ${s_p}^i = [{si_p}^0, {si_p}^1, \cdots, {si_p}^i]$, represent the sequence of all state intervals up to ${si_p}^i$. ${s_p}^i$ and ${s_q}^j(p \neq q)$ 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. A set of states, which consists of only one from each process in the system, is $a\ globally\ consistent\ state$ if any pair of the states is mutually consistent [7].

3 The Proposed 2-step Algorithm

In this section, we describe an efficient algorithm consisting of two steps taken depending on the available empty buffer space for logging messages in future. The proposed algorithm is executed like in figure 2. As the first step, while the free buffer space of each process is larger than its lower bound LB, the process locally removes useless log information from its volatile storage by using only a vector piggybacked on each received message. In this case, the algorithm results in no additional message and forced checkpoint. In case that the free buffer space is smaller than LB in some checkpointing and communication patterns, the second step forces a part of useful log information in the buffer to be useless and removed until the free space becomes the upper bound UB. In this case, the algorithm chooses a minimum number of processes to participate in the garbage collection based on an

array recording the current size of the log information in its buffer for every other process. Therefore, regardless of specific checkpointing and communication patterns, the 2-step algorithm enables the cost of garbage collection performed during failure-free operation to be significantly reduced compared with the existing algorithms. Next subsections will explain the two steps of the algorithm in detail respectively.

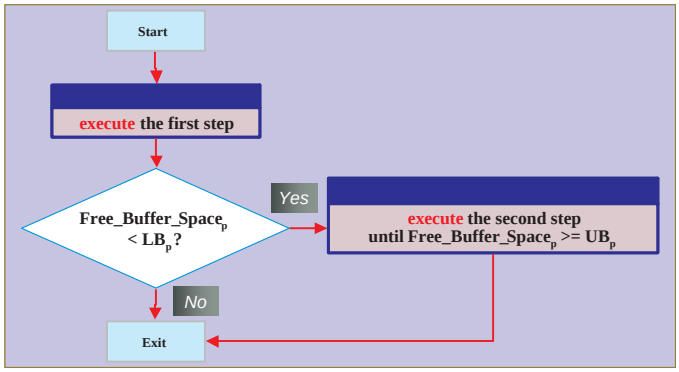

Figure 2: The 2-step algorithm for process p

3.1 The First Step

The sender-based message logging approach has the feature that each failed process has to be rolled back to the latest checkpoint and replay the received messages beyond the checkpoint by obtaining the recovery information from their sender processes. From this feature, we can see that all the messages received before process p takes its latest checkpoint, are useless for recovering p to a consistent state in case of p's failure. For example, there are three processes $p1$, $p2$ and $p3$ in figure 3. In here, process $p1$ sends two messages $msg1$ and $msg3$ to $p2$ after having saved the log information of the messages in its volatile storage. Also, process $p3$ sends message $msg2$ to $p2$ in the same way. In this case, we suppose that process $p2$ takes its i-th checkpoint after it received the three messages like in this figure. Afterwards, even if $p2$ fails, it rolls back at most up to the i-th checkpoint. Thus, the log information of the three messages $msg1$, $msg2$ and $msg3$ becomes useless in case of future failures.

Therefore, the first step of the proposed algorithm is designed to enable a process p to locally remove the useless logged messages from the volatile storage without requiring any extra message and forced checkpoint. For this purpose, each process p must have the following data structures.

- $Sendlg_p$: a set saving $lge(rid, ssn, rsn, data)$ of each message sent by p. In here, lge is the log information of a message and the four fields are the identifier of the receiver, the send sequence number, the receive sequence number and data of the message respectively.

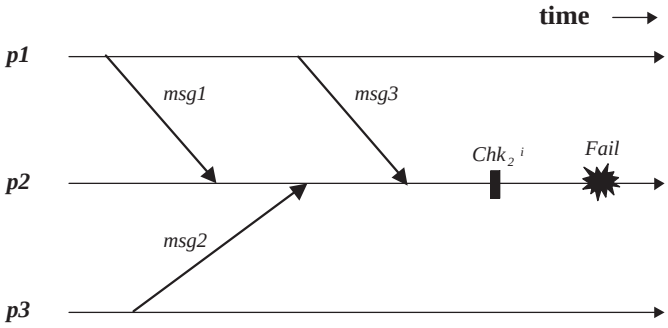

Figure 3: What is useless log information in sender based message logging?

- Rsn_p: the receive sequence number of the latest message delivered to p.
- Ssn_p: the send sequence number of the latest message sent by p.
- $SsnVector_p$: a vector in which $SsnVector_p[q]$ records the send sequence number of the latest message received by p that q sent.
- $RsnVector_p$: a vector in which $RsnVector_p[k]$ is the receive sequence number of the last message delivered to k before k has saved the last checkpointed state of k on the stable storage.
- $EnableS_p$: a set of rsns that aren't yet recorded at the senders of their messages. It is used for indicating whether p can send messages to other processes.

Informally, our algorithm is performed as follows. Taking a local checkpoint, p updates $RsnVector_p[p]$ to the receive sequence number of the latest message delivered to p. If p sends a message m to another process q, the vector is piggybacked on the message. When receiving the message with $RsnVector_p$, q takes the component-wise maximum of two vectors $RsnVector_p$ and $RsnVector_q$. Afterwards, q can remove from its message log $Sendlg_q$ all $lge(u)$s such that for all $k \in$ a set of all processes in the system, $lge(u).rid$ is k and $lge(u).rsn$ is less than or equal to $RsnVector_p[k]$.

To explain the algorithm more easily, figure 4 shows an example of a distributed computation consisting of three processes $p1$, $p2$ and $p3$ communi-

cating with each other. In this example, the processes take their local checkpoints Chk_1^w, Chk_2^x and Chk_3^y. In this case, they update $RsnVector_1[1]$, $RsnVector_2[2]$ and $RsnVector_3[3]$ to each rsn of the last message received before taking its respective checkpoint. In here, we assume that values of Rsn_1, Rsn_2 and Rsn_3 are a, b and c. Afterwards, $p2$ receives four messages, $msg1$ and $msg5$ from $p1$ and $msg2$ and $msg4$ from $p3$. At this point, $p1$ keeps $lge(msg1)$ and $lge(msg5)$ in $Sendlg_1$, and $p3$, $lge(msg2)$ and $lge(msg4)$ in $Sendlg_3$. On taking the next local checkpoint Chk_2^{x+1}, $p2$ updates $RsnVector_2[2]$ to rsn of $msg5$ as $msg5$ is the last message received before the checkpoint. In this case, the value of Rsn_2 becomes (b+4). Then, it sends a message $msg7$ with $RsnVector_2$ to $p1$. When receiving the message, $p1$ updates $RsnVector_1[2]$ to (b+4). Thus, it can remove useless log information, $lge(msg1)$ and $lge(msg5)$, from $Sendlg_1$ because rsn of message $msg5$ is equal to $RsnVector_1[2]$. Hereafter, it takes the next local checkpoint Chk_1^{w+1} and so sets the value of $RsnVector_1[1]$ to rsn of the last message, $msg7$, received before taking the checkpoint. In this case, $RsnVector_1[1]$ becomes (a+3). After that, it sends a message $msg8$ with $RsnVector_1$ to $p3$. On receiving the message, $p3$ updates $RsnVector_3$ to (a+3, b+4, c) by using the vector of $p1$ piggybacked on the message. It can remove $lge(msg2)$, $lge(msg4)$ and $lge(msg6)$ from $Sendlg_3$ because rsns of messages $msg4$ and $msg6$ are less than $RsnVector_3[2]$ and $RsnVector_3[1]$ respectively. Then, $p3$ sends $p2$ a message $msg9$ with $RsnVector_3$. When $p2$ receives the message, $RsnVector_2$ becomes (a+3, b+4, c) after updating it. In this case, $p2$ can remove useless $lge(msg3)$ and $lge(msg7)$ from $Sendlg_2$. From this example, we can see that the algorithm allows each process to locally remove useless log information from its volatile storage with no extra messages and forced checkpoints.

However, in some checkpointing and communication patterns like figure 5, the first step cannot allow each process to autonomously decide whether log information of each sent message is useless for recovery of the receiver of the message by using some piggybacking information. In the traditional sender-based message logging protocols, to garbage collect every $lge(m)$ in $Sendlg_p$, p requests that the receiver of m ($m.rid$) takes a checkpoint if it has indeed received m and taken no checkpoint since. Also, processes occasionally exchange the state interval indexes of their most recent checkpoints for garbage collecting the log information in their volatile storages. However, the previous algorithm may result in a large number of additional messages and forced checkpoints needed by the forced garbage collection.

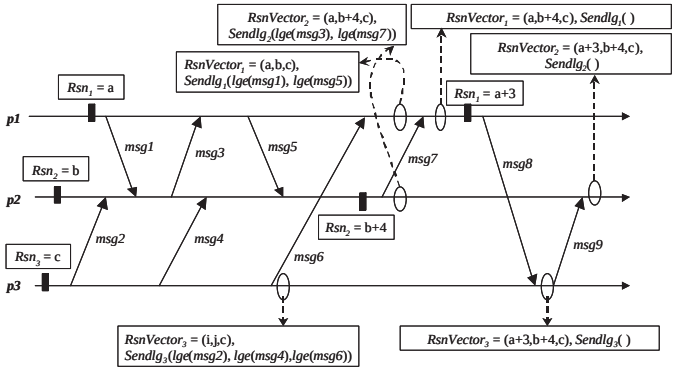

Figure 4: An example showing the effectiveness of the first step of the proposed algorithm

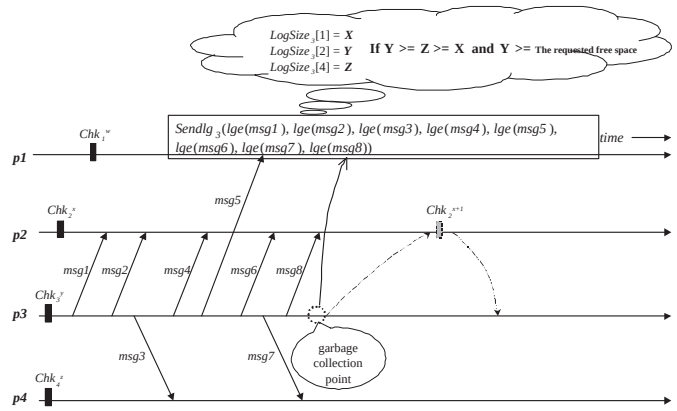

Figure 5: An example of executing the second step of the proposed algorithm

To illustrate how to remove the log information in the algorithm, consider the example shown in figure 5. Suppose $p3$ intends to remove the log information in $Sendlg_3$ at the marked point. In this case, the algorithm forces $p3$ to send checkpoint requests to $p1$, $p2$ and $p4$. When receiving the request, $p1$, $p2$ and $p4$ take their checkpoints respectively. Then, the three processes send each a checkpoint reply to $p3$. After receiving all the replies, $p3$ can remove $(lge(msg1), lge(msg2), lge(msg3), lge(msg4), lge(msg5), lge(msg6), lge(msg7), lge(msg8))$ from $Sendlg_3$. Thus, the previous algorithm results in high synchronization overhead.

3.2 The Second Step

To solve the problem mentioned earlier, the second step of the proposed algorithm is designed based on the following observation: if the requested empty space (=E) is less than or equal to the sum

(=Y) of sizes of $lge(msg1)$, $lge(msg2)$, $lge(msg4)$, $lge(msg6)$ and $lge(msg8)$, $p3$ has only to force $p2$ to take a checkpoint. This observation implies that the number of extra messages and forced checkpoints may be reduced if $p3$ knows sizes of the respective log information for $p1$, $p2$ and $p4$ in its volatile storage. The second step obtains such information by maintaining an array, $LogSize_p$, to save the size of the log information in the volatile storage by process. Thus, the algorithm can reduce the number of additional messages and forced checkpoints by using the vector compared with the traditional algorithm.

The second step needs a vector $LogSize_p$ where $LogSize_p[q]$ is the sum of sizes of all $lge(m)$s in $Sendlg_p$, such that p sent message m to q. Whenever p sends m to q, it increments $LogSize_p$ by the size of $lge(m)$. When p needs more empty buffer space, it executes the second step of the algorithm. It first chooses a set of processes, denoted by $participatingProcs$, which will participate in the forced garbage collection. It selects the largest, $LogSize_p[q]$, among the remaining elements of $LogSize_p$, and then appends q to $participatingProcs$ until the required buffer size is satisfied. Then p sends a request message with the rsn of the last message, sent from p to q, to all $q \in participatingProc$ such that the receiver of m is q for $\exists lge(m) \in Sendlg_p$. When q receives the request message with the rsn from p, it checks whether the rsn is greater than $RsnVector_q[q]$. If so, it should take a checkpoint and then send p a reply message including $RsnVector_q[q]$. Otherwise, it has only to send p a reply message. When p receives the reply message from q, it removes all $lge(m)$s from $Sendlg_p$ such that the receiver of m is q.

For example, in figure 5, when $p3$ attempts to execute the second step at the marked point after it has sent $msg8$ to $p2$, it should create $participatingProcs$. In this figure, we can see that $LogSize_3[2]$(= Y) is the largest ($Y \geq Z \geq X$) among all the elements of $LogSize_3$ due to $lge(msg1)$, $lge(msg2)$, $lge(msg4)$, $lge(msg6)$ and $lge(msg8)$ in $Sendlg_3$. Thus, it first selects and appends $p2$ to $participatingProcs$. Suppose that the requested empty space E is less than or equal to Y. In this case, it needs to select any process like $p1$ and $p4$ no longer. Therefore, $p3$ sends a checkpoint request message with $msg8.rsn$ only to $p2$ in $participatingProcs$. When $p2$ receives the request message, it should take a forced checkpoint like in this figure because the rsn included in the message is greater than $RsnVector_2[2]$. Then it sends $p3$ a reply. When $p3$ receives a reply message from $p2$, it can remove $lge(msg1)$, $lge(msg2)$, $lge(msg4)$, $lge(msg6)$ and $lge(msg8)$ from $Sendlg_3$. From this example, we can see that the second step chooses a small number of processes to participate in the garbage collection based on $LogSize_3$ compared with the traditional algorithm. Thus, this algorithm may reduce the number of additional messages and forced checkpoints.

3.3 Algorithmic Description

The procedures for process p in our algorithm are formally described in figures 6 and 7. MSGS() in figure 6 is the procedure executed when each process p sends a message m and logs the message to its volatile memory. In this case, p piggybacks $RsnVector_p$ on the message for the first step of the algorithm and then adds the size of $e(m)$ to $LogSize_p$[q] after transmitting the message for the second step. Procedure MSGR() is executed when p receives a message. In this procedure, p first notifies the sender of the message of its rsn and then performs the first step for removing useless log information from its log based on the piggybacked vector. In procedure ACK-RECV(), process p receives the rsn of its previously sent message and updates the third field of the element for the message in its log to the rsn. Then, it confirms fully logging of the message to its receiver, which executes procedure CONFIRM-RECV(). If process p attempts to take a local checkpoint, it calls procedure CHECKPOINTING(). In this procedure, the element for p of $RsnVector_p$ is updated to the rsn of the last message received before the checkpoint. STEP-2 in figure 7 is the procedure executed when each process attempts to initiate the forced garbage collection of the second step and CHECKLRSNINLCHKPT() is the procedure for forcing the log information to become useless for future recovery.

4 Correctness Proof

In this section, we prove the correctness of the first and the second steps of the proposed algorithm.

Lemma 1. If $si_q{}^j$ is created by message m from p to q ($p \neq q$) for all $p, q \in P$ and then q takes its latest checkpoint in $si_q{}^l$ ($l \geq j$), $lge(m)$ need not be maintained in $Sendlg_p$ for q's future recovery in the sender-based message logging.

Proof : We prove this lemma by contradiction. Assume that $lge(m)$ in $Sendlg_p$ is useful for q's future recovery in case of the condition. If q fails, it restarts execution from its latest checkpointed state for its recovery in the sender-based message logging.

procedure MSGS($data$, q)
  wait until($EnableS_p$ = Φ) ;
  $Ssn_p \leftarrow Ssn_p$ + 1 ;
  send $m(Ssn_p, data)$ with $RsnVector_p$ **to** q ;
  $Sendlg_p \leftarrow Sendlg_p \cup \{(q, Ssn_p, \text{-1}, data)\}$;
  $LogSize_p[q] \leftarrow LogSize_p[q]$ + size of (q, Ssn_p, -1, $data$) ;

procedure MSGR($m(ssn, data)$, sid, $RsnVector$)
  if($SsnVector_p[sid] < m.ssn$) **then** {
    $Rsn_p \leftarrow Rsn_p$ + 1 ;
    $SsnVector_p[sid] = m.ssn$;
    send $ack(m.ssn, Rsn_p)$ **to** sid ;
    $EnableS_p \leftarrow EnableS_p \cup \{Rsn_p\}$;
    for all $k \in$ other processes in the system **do**
     if($RsnVector_p[k] < RsnVector[k]$) **then** {
       $RsnVector_p[k] \leftarrow RsnVector[k]$;
       for all $e \in Sendlg_p$ **st** $((e.rid = k) \wedge$
           $(e.rsn \leq RsnVector[k]))$ **do**
           $Sendlg_p \leftarrow Sendlg_p$ - $\{e\}$;
     }
     deliver $m.data$ **to** the application ;
  } **else discard** m ;

procedure ACK-RECV($ack(ssn, rsn)$, rid)
  find $\exists e \in Sendlg_p$ **st** $((e.rid = rid) \wedge$
    $(e.ssn = ack.ssn))$;
  $e.rsn \leftarrow ack.rsn$;
  send $confirm(ack.rsn)$ **to** rid ;

procedure CONFIRM-RECV($confirm(rsn)$)
  $EnableS_p \leftarrow EnableS_p$ - $\{rsn\}$;

procedure CHECKPOINTING()
  $RsnVector_p[p] \leftarrow Rsn_p$;
  take its local checkpoint **on** the stable storage ;

Figure 6: Procedures for every process p in algorithm 2-*step*(continued)

procedure STEP-2($sizeOflogSpace$)
  $participatingProcs \leftarrow \emptyset$;
  while $sizeOflogSpace > 0$ **do**
    if(there is r **st** $((r \in P) \wedge (r$ is not an element of $participatingProcs) \wedge (LogSize_p[r] \neq 0)$ $\wedge$ (**max** $LogSize_p[r]$))) **then** {
      $sizeOflogSpace \leftarrow sizeOflogSpace -$ $LogSize_p[r]$;
      $participatingProcs \leftarrow$ $participatingProcs \cup \{r\}$;
    }
  T: **for all** $u \in participatingProcs$ **do** {
      $MaximumRsn \leftarrow$ (**max** $e(m).rsn$) **st** $((e(m) \in Sendlg_p) \wedge (u = e(m).rid))$;
      send $Request(MaximumRsn)$ **to** u ;
    }
    while $participatingProcs \neq \emptyset$ **do** {
      receive $Reply(rsn)$ **from** u **st** $(u \in participatingProcs)$;
      for all $e(m) \in Sendlg_p$ **st** $(u = e(m).rid)$ **do**
        remove $e(m)$ **from** $Sendlg_p$;
      $LogSize_p[u] \leftarrow 0$;
      $participatingProcs \leftarrow participatingProcs$ $-\{u\}$;
    }

procedure CHECKLRSNINLCHKPT($Request($ $MaximumRsn), q$)
  if($RsnVector_p[p] < MaximumRsn$) **then**
    CHECKPOINTING() ;
  send $Reply(RsnVector_p[p])$ **to** q ;

Figure 7: Procedures for every process p in algorithm 2-*step*

In this case, p need not retransmit m to q because $dev_q(m)$ occurs before the checkpointed state. Thus, $lge(m)$ in $Sendlg_p$ is not useful for q's recovery. This contradicts the hypothesis. □

Theorem 1. The first step of the proposed algorithm removes only the log information that will not be used for future recoveries in sender-based message logging any longer.

Proof : Let us prove this theorem by contradiction. Assume that our algorithm removes the log information useful for future recoveries. As mentioned in section 3.1, the algorithm forces each process p to remove log information from its volatile memory only in the following case.

Case 1: p receives a message m from another process q (i.e., when executing **procedure** MSGR() in figure 6).
In this case, $RsnVector_p$ was piggybacked on m. Thus, p removes from $Sendlg_p$ all $lge(l)$s such that for $k \in P$ ($k \neq p$), $lge(l).rid$ is k and $lge(l).rsn$ is less than or equal to **max**($RsnVector_p[k]$, $RsnVector_q[k]$), which is the rsn of the last message delivered to k before having taken its latest checkpoint. In here, message l need no longer be replayed in case of failures of process k due to its latest checkpoint. Thus, $lge(l)$ isn't useful for its future recoveries.

Therefore, The first step of the proposed algorithm removes only useless log information for sender-based message logging in any case. This contradicts the hypothesis. □

Theorem 2. After every process has performed the second step of the proposed algorithm in the sender-based message logging, the system can recover to a globally consistent state despite process failures.

Proof : the second step of the proposed algorithm only removes the following useful log information in the storage buffer of every process as follows.

Case 1: Process p for all $p \in P$ removes any $lge(m)$ in $Sendlg_p$.
In this case, it sends a request message with the rsn of the last message, sent from p to $lge(m).rid$, to $lge(m).rid$. When $lge(m).rid$ receives the request message with the rsn from p, it checks whether the rsn is greater than $RsnVector_{lge(m).rid}[lge(m).rid]$.

Case 1.1: The rsn is greater than $RsnVector_{lge(m).rid}[lge(m).rid]$.
In this case, $lge(m).rid$ takes a checkpoint. Afterwards, $lge(m)$ becomes useless for the sender-based message logging by lemma 1.

Case 1.2: The rsn is less than or equal to $RsnVector_{lge(m).rid}[lge(m).rid]$.
In this case, $lge(m).rid$ took its latest checkpoint after having received m. Thus, $lge(m)$ is useless for the sender-based message logging by lemma 1.

Thus, all the useful log information for the sender-based message logging is always maintained in the system in all cases. Therefore, after every process has performed the second step of the proposed algorithm, the system can recover to a globally consistent state despite process failures. □

5 Performance Evaluation

5.1 Simulation Environment

To evaluate performance of our algorithm(2-$step$) with that of the traditional one ($Tradi$) [12], some experiments are performed in this paper using a discrete-event simulation language [19]. First, one performance index is used for evaluating the effectiveness of the first step of the proposed algorithm; the average elapsed time required until the volatile memory buffer for message logging of a process is full(T_{full}). The performance index T_{full} is measured under the condition that the two algorithms perform no forced garbage collection procedure, i.e., incur no additional messages and no forced checkpoints. Second, the following performance indexes are used for comparing forced garbage collection overheads of both the second step of algorithm 2-$step$ and algorithm $Tradi$; the average number of additional messages ($NOAM$) and the average number of forced checkpoints ($NOFC$) required for garbage collection per process.

A simulated system consists of 20 hosts connected by a network, which is modelled as a multi-access LAN (Ethernet). The message transmission capacity of a link in the network is 100 Mbps. Nodes connected to the network are identical and uniformly distributed along the physical medium. For simplicity of this simulation, it is assumed each node has one process executing on it and 20 processes are initiated and completed together. For the experiments, it is also assumed that the size of each application message ranges from 50 to 200 Kbytes and the size of the memory buffer for logging of every process is 10Mbytes. Each process takes its local checkpoint with an interval following an exponential distribution with a mean

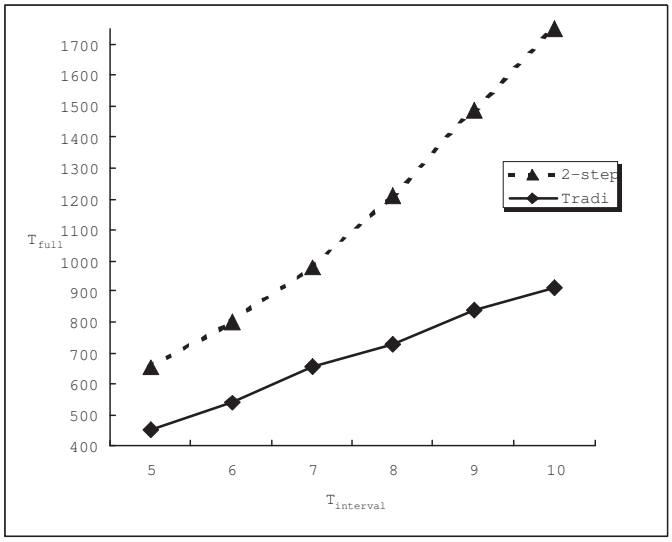

Figure 8: Average elapsed time required until the volatile memory buffer for message logging of a process is full according to $T_{interval}$

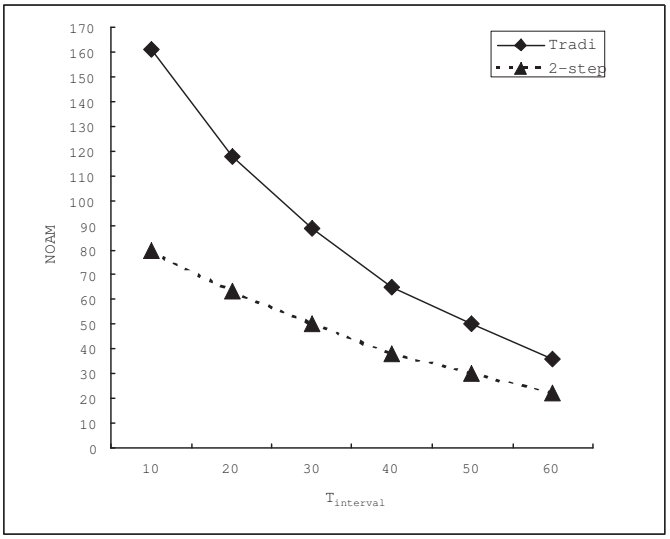

Figure 9: $NOAM$ vs. $T_{interval}$

$Ckpt_{time}$=3 minutes. The simulation parameter is the mean message sending rate, $T_{interval}$, following an exponential distribution. All simulation results shown in this section are averages over a number of trials.

5.2 Simulation Results

Figure 8 shows the average elapsed time of the two algorithms required until the volatile memory buffer for message logging of a process is full for the specified range of the $T_{interval}$ values. In this figure, as their $T_{interval}$s of algorithms 2-$step$ and $Tradi$ increase, their corresponding T_{full}s also increase. The reason is that as each process sends messages more slowly, the size of its message log also increases at a lower rate. However, as it is expected, T_{full} of algorithm 2-$step$ is significantly higher than that of algorithm $Tradi$. In particular, as $T_{interval}$ increases, the increasing rate of the first rises more fast than that of the latter. This benefit of our algorithm results from its desirable feature as follows: it enables a process p to autonomously and locally eliminate useless log information from the buffer by only carrying a vector $RsnVector_p$ on each sent message whereas the traditional algorithm does not so.

Figure 9 shows $NOAM$ for the various $T_{interval}$ values. In this figure, we can see that $NOAM$s of the two algorithms increase as $T_{interval}$ decreases. The reason is that forced garbage collection should frequently be performed because the high inter-process communication rate causes the storage buffer of each process to be overloaded quickly. However, $NOAM$ of algorithm 2-$step$ is much lower than that of algorithm $Tradi$. Algorithm 2-$step$ reduces about 38% - 50% of $NOAM$ compared with algorithm $Tradi$.

Figure 10 illustrates $NOFC$ for the various $T_{interval}$ values. In this figure, we can also see that $NOFC$s of the two algorithms increase as $T_{interval}$ decreases. The reason is that as the inter-process communication rate increases, a process may take a forced checkpoint when it performs forced garbage collection. In the figure, $NOFC$ of algorithm 2-$step$ is lower than that of algorithm $Tradi$. Algorithm 2-$step$ reduces about 25% - 51% of $NOFC$ compared with algorithm $Tradi$.

Therefore, we can conclude from the simulation results that regardless of the specific checkpointing and communication patterns, algorithm 2-$step$ enables the garbage collection overhead occurring during failure-free operation to be significantly reduced compared with algorithm $Tradi$.

6 Conclusion

This paper presents a novel log management algorithm to effectively eliminate the volatile log information at sender processes on demand without the violation of the system consistency. First, the algorithm gets rid of needless logged messages from the corresponding senders' volatile memories only by piggybacking a vector on their sent messages. This advantageous feature results in no additional message and forced checkpoint. If the more empty buffer space for the volatile logging is needed after the first pro-

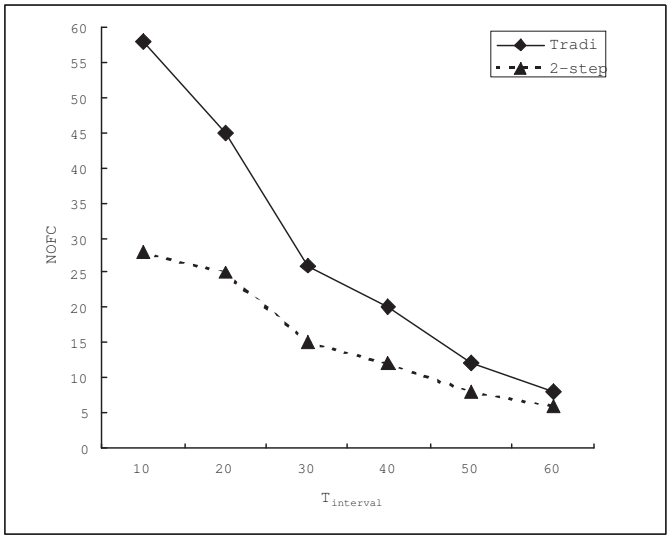

Figure 10: $NOFC$ vs. $T_{interval}$

cedure executed, the next procedure of this proposed algorithm is performed to address this limitation. This procedure uses a vector for saving the size of the log information required to recover every other process and enables the information to be efficiently removed while satisfying the consistency condition.

Acknowledgements: This work was supported by Gyeonggido Regional Research Center Program grant(2007-081-2, Development and Industrialization of Integrated Frameworks for Very Large-scale RFID Services).

References:

[1] A. Acharya and B. R. Badrinath. Checkpointing Distributed Applications on Mobile Computers. *In Proc. the 3th International Conference on Parallel and Distributed Information Systems*, 1994.

[2] J. Ahn. Lightweight Fault-tolerant Message Passing System for Parallel and Distributed Applications. *Lecture Series on Computer and Computational Sciences*, Vol. 8, pp. 12-15, Oct. 2008.

[3] J. Ahn. Effective Service Replication Mechanisms Exploiting Agent Mobility. *Proc. of the 7th WSEAS International Conference on Software Enineering, Parallel and Distributed Systems*, pp. 74-79, Feb. 2008.

[4] F. Baschieri, P. Bellavista and A. Corradi. Mobile Agents for Qos Tailoring, Control and Adaptation over the Internet: The UbiQoS Video on Demand Serbvice. *In Proc. of the 2nd International Symposium on Applications and the Internet*, pp. 109-118, 2002.

[5] A. Bouteiller, F. Cappello, T. Hérault, 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 15th International Conference on High Performance Networking and Computing(SC2003)*, November 2003.

[6] H. Bryhni, E. Klovning and O. Kure. A Comparison of Load Balancing Techniques for Scalable Web Servers. *IEEE Network*, 14:58-64, 2000.

[7] K. M. Chandy, and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1): 63-75, 1985.

[8] Y. Chen and D. Li. A Web-GIS based Decision Support System for Revegetation in Coal Mine Waste Land. *Proc. of the 7th WSEAS International Conference on Applied Computer and Applied Computational Science*, pp. 579-584, April 2008.

[9] J. Cui and H. Chae. Mobile Agent based Load Balancing for RFID Middlewares. *In Proc. of International Conference on Advanced Computer Technology*, pp. 973-978, 2007.

[10] C. Curino, M. Giani, M. Giorgetta, A. Giusti, A. Murphy and G. Picco. Mobile Data Collection in Sensor Networks: The TinyLime Middleware. *Journal of Pervasive and Mobile Computing*, 4(1):446-469, December 2005.

[11] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel. The Performance of Consistent Checkpointing. *In Proc. the 11th Symposium On Reliable Distributed Systems*, pp. 86-95, 1992.

[12] D. B. Johnson and W. Zwaenpoel. Sender-Based Message Logging. *In Digest of Papers: 17th International Symposium on Fault-Tolerant Computing*, pp. 14-19, 1987.

[13] 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), pp. 375-408, 2002.

[14] J. L. Kim and T. Park. An Efficient Protocol For Checkpointing Recovery in Distributed Systems. *IEEE Transactions on Parallel and Distributed Systems*, pp. 955-960, 1993.

[15] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*. Vol.13, No.1, pp. 23-31, 1987.

[16] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21, pp. 558-565, 1978.

[17] Z. Li and M. Parashar. A Decentralized Agent Framework for Dynamic Composition and Coordination for Autonomic Applications. *Proc. of the 3rd International Workshop on Self-Adaptive and Autonomic Computing Systems*, Copenhagen, Denmark, pp. 165-169, August 2005.

[18] D. Manivannan and Mukesh Singhal. A Low-Overhead Recovery Technique Using Quasi-Synchronous Checkpointing. *In Proc. the 16th International Conference on Distributed Computing Systems*, pp. 100-107, 1996.

[19] R. McNab and F. W. Howell. simjava: a discrete event simulation package for Java with applications in computer systems modelling. *In Proc. First International Conference on Web-based Modelling and Simulation*, 1998.

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

[21] J. T. Rough and A. M. Goscinski. The development of an efficient checkpointing facility exploiting operating systems services of the GENESIS cluster operating system. *Future Generation Computer Systems*, Vol. 20, No. 4, pp 523-538, 2004.

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

[23] M. O. Spata, S. Rinaudo, A. Marotta and F. Moschella. Integration of a Parallel Algorithm with a Cluster Grid for an Industrial Framework. *Proc. of the 7th WSEAS International Conference on Software Enineering, Parallel and Distributed Systems*, pp. 27-31, Feb. 2008.

[24] R.E. Strom and S.A. Yemeni. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3, pp. 204-226, 1985.

[25] 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*, pp. 602-609, 1995.

[26] B. Yao, K. -F. Ssu and W. K. Fuchs. Message Logging in Mobile Computing. *In Proc. of the 29th International Symposium on Fault-Tolerant Computing*, pp. 14-19, 1999.