Algorithm for Reducing Overhead of Message Purging in Large-Scale Sender-Based Logging

Jinho Ahn

Dept. of Computer Science, College of Natural Science, Kyonggi University
Suwon, Gyeonggi-do 443-760, Republic of Korea

Abstract

When attempting to apply sender-based message logging with checkpointing into large-scale and geographically distributed systems, two important things should be reconsidered: reducing the number of messages passing on core networks during its fully message logging and recovery procedures and purging effectively logged messages from their senders' volatile memories. This paper presents a novel message purging algorithm to solve the second problem based on our previous work having alleviated the first one. Its first step results in no extra message and forced checkpoint by piggybacking a vector on each sent message. If additional empty buffer space for logging is needed even after the first step has executed, its second step is performed to remove the useful log information each sender maintains while satisfying the consistency condition by using a vector recording its size for every other process.

Keywords: *distributed systems, roll-back recovery, message logging, checkpointing, message purging.*

1. Introduction

Thanks to highly rapid advances in processor and network technologies, a lot of architectural variations in distributed computing systems are being experienced and reflected on many application fields such as p2p computing, collaborative computing, ubiquitous sensor networks, network monitoring, grid and cloud computing and so on. But, as their collaborative feature may cause themselves to be more vulnerable to process failures, the systems require some cost-effective fault-tolerance techniques [8]. Log-based rollback recovery is such a technique in that each process periodically saves it local state by or without synchronizing with other processes [3], [9], [10] and logs each received message [5]. 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 [5]. First, receiver-based message logging approach [11], [15] logs the recovery information of every received message into the stable storage before the message is delivered to its 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 [2], [6], [14] enables each message to be logged in the volatile memory of its corresponding sender for avoiding logging messages into the stable storage synchronously. Therefore, it significantly 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 their receivers when they crash.

As architectural aspects of current and future distributed computing systems are changing to geographically group-based and peer-to-peer based, many of these systems are adopting representative-based architecture like broker-based sensor networks, super-peer based P2P systems, etc., to accommodate these topological features well. Thus, this change is making a lot of issues about their fundamental building blocks reconsidered to work well for these newly fashioned systems in highly effective manners. Sender-based message logging abbreviated by SBML should also be examined properly before its application to accommodate this architectural change. Two important things should be reconsidered:

- Be reducing the number of messages passing on core networks during its fully message logging and recovery procedures
- Purging effectively logged messages from their senders' volatile memories.

In our previous work [1], the first drawback was alleviated to enable the representative elected in a cluster or group of nodes like broker or super-peer to localize both of the logging and recovery procedures to a maximum. This

ISSN (Online): 1694-0814
www.IJCSI.org

paper presents a novel message purging algorithm to efficiently remove logged messages from the volatile storage while ensuring the consistent recovery of the system in case of node 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 additional empty buffer space for logging messages in future may be required. In this case, the second step of our algorithm forces some useful log information in each sender's volatile memory to become useless by maintaining a vector recording the size of the information for every other process' failure. This algorithm can choose a minimum number of processes participating in the message purging procedure based on the vector. Thus, this behavior incurs fewer additional messages and forced checkpoints than the existing ones.

The remainder of the paper is as follows. Section 2 describes the distributed computing system model assumed in this paper and section 3 introduces our message purging algorithm in details. In sections 4 and 5, we prove the correctness of the algorithm and evaluate performance of our work and the representative previous work in several aspects. Section 6 summarizes our work.

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[5]. 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 are reliably 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 [12]. Events of processes occurring in a failure-free execution are ordered using Lamport's happened before relation [7]. The execution of each process is piecewise deterministic [13]: 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, $[dev_p^1, dev_p^2, \ldots, dev_p^i]$, its corresponding state s_p^i is uniquely determined. Let p's state, $s_p^i = [si_p^0, si_p^1, \ldots, si_p^i]$, 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 [4]. 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. In the remainder of this paper, the messages applications generate are called application messages and the messages used for the message logging and recovery procedures, control messages.

3. The Proposed 2-step Message Purging Algorithm

3.1 Basic Concepts

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 as follows. 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 message purging procedure based on an array recording the current size of the log information in its buffer for every other process. Therefore, regardless of particular checkpointing and communication patterns, the 2-step algorithm enables the cost of the message purging procedure performed during failure-free operation to be significantly reduced compared with the existing algorithms.

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 checkpoints. For this purpose, each process p must have the following data structures.

- $Sendlg_p$: a set saving e(rid, ssn, rsn, data) of each message sent by p. In here, e 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.
- Rsn_p: the receive sequence number of the latest message delivered to p.
- Ssn_p: the send sequence number of the latest message

IJCSI
www.IJCSI.org

ISSN (Online): 1694-0814
www.IJCSI.org

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 e(u)s such that for all k $\in$ a set of all processes in the system, e(u).rid is k and e(u).rsn is less than or equal to $RsnVector_p[k]$. However, in some checkpointing and communication patterns, the first step may not 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 purge away each e(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 purging the log information from their volatile storages. However, the previous algorithm may result in a large number of additional messages and forced checkpoints needed by the forced purging procedure in an explicit manner.
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 e(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 e(m). When p needs additional 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 purging procedure. 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 $\exists$e(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 the reply message. When p receives the reply message from q, it removes all e(m)s from $Sendlg_p$ such that the receiver of m is q.

3.2 Algorithmic Description

The procedures for process p in our algorithm are formally described in figures 1 and 2. MSG-SEND() in figure 1 is the procedure executed when each process p sends a message m to its receiver or the representative of the receiver and logs the message into 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. In procedure MSG-RECV1(), the following two parts are performed according to who its final receiver is. If the process receiving the message is its final receiver, it invokes procedure MSG-RECV2(). Otherwise, it forwards the message to its final destination and then performs the first step by calling procedure MAX-RSNS&PURGE-MSGS(). Next, it logs the message into its volatile memory and then adds the size of e(m) to $LogSize_{Ai}[rcvr]$. Procedure MSG-RECV2() 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 RSN-RCVR(), 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 RSN-CONFIRM(). In this case, it purges useless log information from its volatile storage by using RsnVector piggybacked on the message. 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. STEP2() in figure 2 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.

Module MSG-SEND(data) OF SENDER P_{sndr}
wait until ($EnableS_{sndr} = \Phi$) ;
$Ssn_{sndr} \leftarrow Ssn_{sndr} + 1$;
if(data is destined to a process P_{rcvr} in another area not playing the role of area representative) **then**
send m(Ssn_{sndr}, data) with $RsnVector_{sndr}$ **to** the representative of P_{rcvr} ;
else send m(Ssn_{sndr}, data) with $RsnVector_{sndr}$ **to** P_{rcvr} ;
$Sendlg_{sndr} \leftarrow Sendlg_{sndr} \cup \{(rcvr, Ssn_{sndr}, -1, data)\}$;
$LogSize_{sndr}[rcvr] \leftarrow LogSize_{sndr}[rcvr]$ + **size of** (rcvr, Ssn_{sndr}, -1, data) ;

Module MSG-RECV1(m, RsnVector) OF Representative R_{Ai}
if(m is a message destined to another process P_{rcvr} in its managing area) **then**
send m with RsnVector **to** P_{rcvr} ;
call MAX-RSNS&PURGE-MSGS(Ai, RsnVector) **at** itself ;
$Sendlg_{Ai} \leftarrow Sendlg_{Ai} \cup \{(rcvr, m.ssn, -1, m.data)\}$;
$LogSize_{Ai}[rcvr] \leftarrow LogSize_{Ai}[rcvr]$ + **size of** (rcvr, m.ssn, -1, m.data) ;
else call MSG-RECV2(m, RsnVector) **at** itself ;

Module MSG-RECV2(m(sid, ssn, data), RsnVector) OF RECEIVER P_{rcvr}
if($SsnVector_{rcvr}[m.sid] < m.ssn$) **then**
$Rsn_{rcvr} \leftarrow Rsn_{rcvr} + 1$;
$SsnVector_{rcvr}[m.sid] \leftarrow m.ssn$;
$EnableS_{rcvr} \leftarrow EnableS_{rcvr} \cup \{(Rsn_{rcvr})\}$;
if(m is a message sent directly from its original sender $P_{m.sid}$) **then**
send ack(m.ssn, Rsn_{rcvr}, $RsnVector_{rcvr}$) **to** m.sid ;
else send ack(m.ssn, Rsn_{rcvr}, $RsnVector_{rcvr}$) **to** the representative of P_{rcvr} ;
call MAX-RSNS&PURGE-MSGS(rcvr, RsnVector) **at** itself ;
deliver m.data **to** its corresponding application ;
else discard m ;

Module RSN-RCVR(ack(ssn, rsn, rid, RsnVector)) OF PROCESS P
find $\exists e \in Sendlg_P$ **st** ((e.rid = ack.rid) ^ (e.ssn = ack.ssn)) ;
e.rsn ← ack.rsn ;
send confirm(ack.rsn) **to** ack.rid ;
call MAX-RSNS&PURGE-MSGS(P, RsnVector) **at** itself ;

Module MAX-RSNS&PURGE-MSGS(pid, RsnVector)
for all k ∈ other processes in the system **do**
if($RsnVector_{pid}[k] < RsnVector[k]$) **then**
$RsnVector_{pid}[k] \leftarrow RsnVector[k]$;
for all $e \in Sendlg_{pid}$ **st** ((e.rid = k) ^(e.rsn ≤ RsnVector[k])) **do**
$Sendlg_{pid} \leftarrow Sendlg_{pid} - \{e\}$;
$LogSize_{pid}[k] \leftarrow LogSize_{pid}[k]$ - **size of** e ;

Module RSN-CONFIRM(m) OF RECEIVER P_{rcvr}
$EnableS_{rcvr} \leftarrow EnableS_{rcvr} - \{(m.rsn)\}$;

Module CHECKPOINTING() OF RECEIVER P
$RsnVector_P[P] \leftarrow Rsn_P$;
take its local checkpoint **on** the stable storage ;

Fig. 1 Message logging with Step 1 procedures

Module STEP2(sizeOflogSpace)
participatingProcs ← Φ ;
while sizeOflogSpace > 0 **do**
if(there is r **st** ((r ∈ P) ^ (r is not an element of participatingProcs) ^ ($LogSize_p[r] \neq 0$)^ (max $LogSize_p[r]$))) **then**
sizeOflogSpace ← sizeOflogSpace – $LogSize_p[r]$;
participatingProcs ← participatingProcs ∪ {r} ;
T: **for all** u ∈ participatingProcs **do**
MaximumRsn ← (**max** e(m).rsn) **st** ((e(m) ∈ $Sendlg_p$)^(u = e(m).rid)) ;
send Request(MaximumRsn) **to** u ;
while participatingProcs ≠ Φ **do**
receive Reply(rsn) **from** u **st** (u ∈ participatingProcs) ;
for all e(m) ∈ $Sendlg_p$ **st** (u = e(m).rid) **do**
remove e(m) **from** $Sendlg_p$;
$LogSize_p[u] \leftarrow 0$;
participatingProcs ← participatingProcs ∪ {u} ;

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

Fig. 1 Step 2 procedures during failure-free operation

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 $si_q^l (j \leq l)$, $lge(m)$ 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. In this case, p need not retransmit m to q because $dev_q^i(m)$ 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 III.A, 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. In this case, $RsnVector_q$ 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 k has taken its latest checkpoint. In here, message l need no longer be replayed in case of failure of process k because of 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. Even if 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 removes the useful log information in the storage buffer of every process in the following cases.

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*) [6], some experiments are performed in this paper using a discrete-event simulation language. 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 four areas where there are each one representative node and four normal nodes. In each area, five nodes including representative are connected through a 100Mbps wired multi-access LAN. All representatives communicate with each other through a 10Mbps wired WAN. 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 $Ckpt_{time}$=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 3 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 much faster 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 4 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 5 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*.

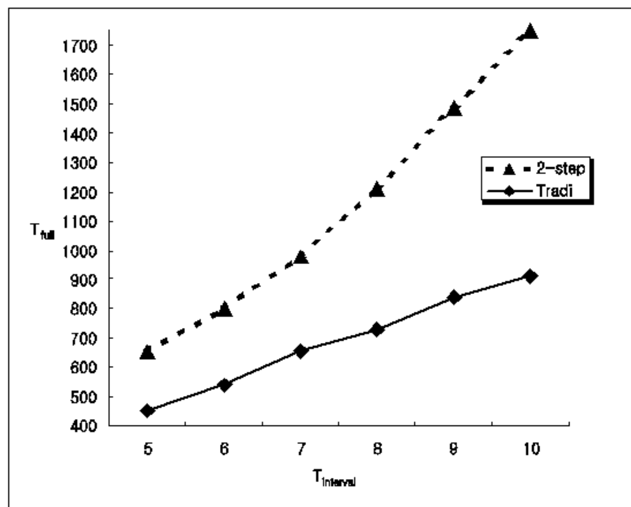

Fig. 3 Average elapsed time required until the volatile memory buffer for message logging of a process is full according to $T_{interval}$

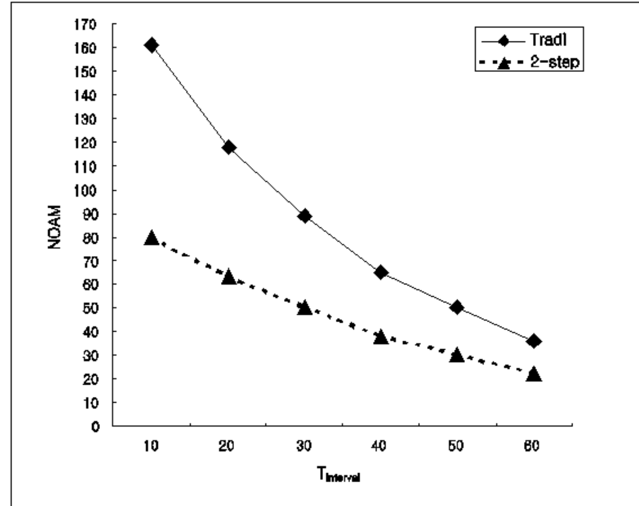

Fig. 4 *NOAM* vs. $T_{interval}$

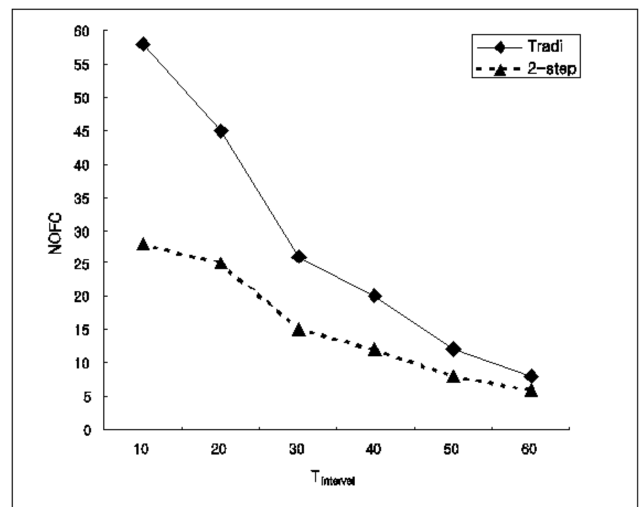

Fig. 5 *NOFC* vs. $T_{interval}$

6. Conclusions

This paper presents a novel message purging algorithm to effectively eliminate the volatile log information at sender processes on demand without the violation of the system consistency. The first step of 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 additional empty buffer space for the volatile logging is needed even after the first step has executed, the next step of this proposed algorithm is performed to address this limitation. This step 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.

References

[1] J. Ahn, "Scalable message logging algorithm for geographically distributed broker-based sensor networks," Proc. of the ISCA 23rd International Conference on Computers and Their Applications in Industry and Engineering (CAINE-2010), 2010, pp. 279-284.

[2] A. Bouteiller, F. Cappello, T. Herault, G. Krawezik, P. Lemarinier, F. Magniette, "MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging," 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, F. Cappello, "Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI Protocols," Future Generation Computer Systems, vol. 24, pp. 73-84, 2008.

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

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

[6] D. Johnson, W. Zwaenpoel, "Sender-based message logging," 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, vol. 21, pp. 558-565, 1978.

[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, vol. 5759, pp. 124-133, 2009.

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

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

[11] M. Powell, D. Presotto, "Publishing: a reliable broadcast communication mechanism," 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, vol. 1, pp. 222-238, 1985.

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

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

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

JINHO AHN(Corresponding author) received his B.S., M.S. and Ph.D. degrees in Computer Science and Engineering from Korea University, Korea, in 1997, 1999 and 2003, respectively. He has been an associate professor in Department of Computer Science, Kyonggi University. He has published more than 70 papers in refereed journals and conference proceedings and served as program or organizing committee member or session chair in several domestic/international conferences and editor-in-chief of journal of Korean Institute of Information Technology and editorial board member of journal of Korean Society for Internet Information. His research interests include distributed computing, fault-tolerance, sensor networks and mobile agent systems.