

# Checkpointing and Communication Pattern-Neutral Algorithm for Removing Messages Logged by Senders

JinHo Ahn

Dept. of Computer Science, College of Science, Kyonggi University  
San 94-6 Iuidong, Yeongtonggu, Suwonsi Kyonggido 443-760, Republic of Korea  
jhahn@kyonggi.ac.kr

**Abstract.** The traditional sender-based message logging protocols use a garbage collection algorithm to result in a large number of additional messages and forced checkpoints. So, in our previous work, an algorithm was introduced to allow each process to autonomously remove useless log information in its volatile storage by piggybacking only some additional information without requiring any extra message and forced checkpoint. However, even after a process has executed the algorithm, its storage buffer may still be overloaded in some communication and checkpointing patterns. This paper proposes a new garbage collection algorithm CCPNA for sender-based message logging to address all the problems mentioned above. The algorithm considerably reduces the number of processes to participate in the garbage collection by using the size of the log information of each process. Thus, CCPNA incurs more additional messages and forced checkpoints than our previous algorithm. However, it can avoid the risk of overloading the storage buffers regardless of the specific checkpointing and communication patterns. Also, CCPNA reduces the number of additional messages and forced checkpoints compared with the traditional algorithm.

**Keyword:** message-passing system, fault-tolerance, message logging, checkpointing, garbage collection.

## 1 Introduction

With the remarkable advance of processor and network technologies, message-passing distributed systems composed of heterogenous networked computers are becoming a cost-effective solution for high performance parallel computing compared with expensive special-purpose supercomputers. However, one of the big challenges the distributed systems should address is providing fault-tolerance. In other words, even if the failure of a single process in a distributed application occurs, it may lead to restarting the application from its initial state, which is critical to long-running scientific and engineering applications. Rollback-recovery techniques such as checkpointing-based recovery and log-based recovery are very attractive for supporting transparent fault-tolerance to the applications because

the techniques require fewer special resources compared to process replication techniques [5]. In checkpointing-based recovery, when some processes crash, the processes affected by the failures roll back to their last checkpoints such that the recovered system state is consistent. But, this technique may not restore the maximum recoverable state because it relies only on checkpoints of processes saved on the stable storage. Log-based recovery performing careful recording of messages received by each process with its checkpoints enables a system to be recovered beyond the most recent consistent set of checkpoints. This feature is desirable for the applications that frequently interact with the outside world consisting of input and output components that cannot roll back [5]. In this technique, messages can be logged either by their senders or by their receivers. First, receiver-based message logging (RBML) approach [8,14] 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 (SBML) approach [2,4,9,11,13] enables each message to be logged in the volatile memory of its corresponding sender for avoiding logging messages to stable storage. Therefore, it reduces the failure-free overhead compared with the RBML approach. However, the SBML 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. Thus, as enough empty buffer space for logging messages sent in future should be ensured in this approach, it requires an efficient algorithm to garbage collect log information of each process [1]. Traditional SBML protocols [2,4,9,11,13] use one between two message log management procedures to ensure system consistency despite future failures according to each cost. The first procedure just flushes the message log 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 procedure forces messages in the log to be useless for future failures and then removes them. In other words, the procedure 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. To address their problems, in our previous work, a low-cost algorithm called PGCA [1] was presented to have the volatile memory of each process for message logging become full as late as possible with no extra message and forced checkpoint. The algorithm allows each process to locally and independently remove useless log information from its volatile storage by piggybacking only some additional information. However, the limitation of the algorithm is that after a process has performed the algorithm, the storage buffer of the process may still be overloaded in some communication and checkpointing patterns. In this paper, we propose an active garbage collection algorithm called CCPNA to lift the limitation. For this, the algorithm CCPNA uses an array recording the size of the log information for each process. When the free buffer

space in the volatile storage is needed, the algorithm selects a small number of processes based on the array that take part in having the messages previously logged for them be useless despite their future failures. Thus, CCPNA results in low communication and checkpointing overheads compared with the traditional ones while avoiding the disadvantage of the algorithm PGCA.

## 2 System Model

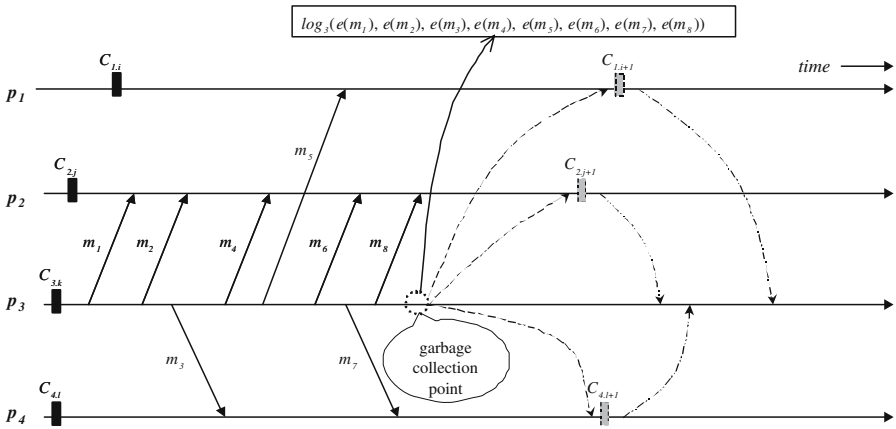
A distributed computation consists of a set  $P$  of  $n(n > 0)$  sequential processes executed on hosts in the system and there is a stable storage that every process can always access that persists beyond processor failures [5]. Processes have no global memory and global clock. The system is asynchronous: each process is executed at its own speed and communicates with each other only through messages at finite but arbitrary transmission delays. We assume that the communication network is immune to partitioning, there is a stable storage that every process can always access and hosts fail according to the fail stop model [10]. Events of processes occurring in a failure-free execution are ordered using Lamport's *happened before* relation [6]. The execution of each process is *piecewise deterministic* [12]: 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  $dev_p^k(m)$ . Let  $p$ 's state,  $s_p^i = \langle si_p^0, si_p^1, \dots, si_p^i \rangle$ , represent the sequence of all state intervals up to  $si_p^i$ . Therefore, given  $p$ 's initial state,  $s_p^0$ , and the non-deterministic events,  $[dev_p^1, dev_p^2, \dots, dev_p^i]$ , its corresponding state  $s_p^i$  is uniquely determined.  $s_p^i$  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 [3].

The log information of each message kept by its sender consists of four fields, its receiving process' identifier(*rid*), send sequence number(*ssn*), receive sequence number(*rsn*) and data(*data*). In this paper, the log information of message  $m$  and the message log in process  $p$ 's volatile memory are denoted by  $e(m)$  and  $log_p$ .

## 3 The Proposed Algorithm

The sender-based message logging needs an algorithm to allow each process to remove the log information in its volatile storage while ensuring system consistency in case of failures. This algorithm should force the log information to become useless for future recovery to satisfy the goal. In the traditional sender-based message logging protocols, to garbage collect every  $e(m)$  in  $log_p$ ,  $p$  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, this algorithm may result in a large number of additional messages and forced checkpoints needed by the forced garbage collection. To illustrate how to remove the log information in the algorithm, consider the example shown in figure 1. Suppose  $p_3$  intends to remove the log information in  $log_{p_3}$  at the marked point. In this case, the algorithm forces  $p_3$  to send checkpoint requests to  $p_1, p_2$  and  $p_4$ . When receiving the request,  $p_1, p_2$  and  $p_4$  take their checkpoints, respectively. Then, the three processes send each a checkpoint reply to  $p_3$ . After receiving all the replies,  $p_3$  can remove  $(e(m_1), e(m_2), e(m_3), e(m_4), e(m_5), e(m_6), e(m_7), e(m_8)))$  from  $log_{p_3}$ . Also, in this checkpointing and communication pattern, the algorithm proposed in [1] cannot allow  $p_3$  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. Thus, even after executing the algorithm,  $p_3$  should maintain all the log information of the eight messages in  $log_{p_3}$ .

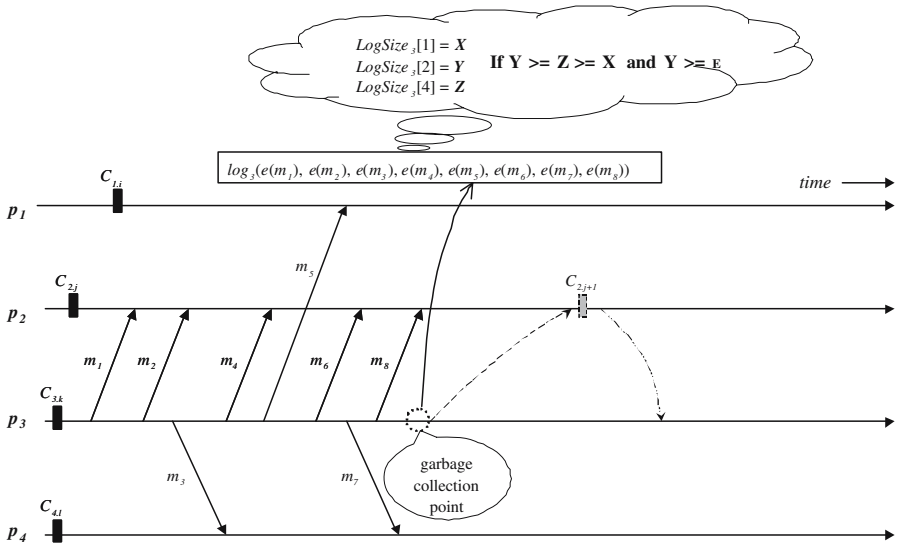


**Fig. 1.** An example showing the problem of the traditional sender-based message logging protocols

To solve the problem, we present an algorithm CCPNA based on the following observation: if the requested empty space ( $=E$ ) is less than or equal to the sum ( $=Y$ ) of sizes of  $e(m_1), e(m_2), e(m_4), e(m_6)$  and  $e(m_8)$ ,  $p_3$  has only to force  $p_2$  to take a checkpoint. This observation implies that the number of extra messages and forced checkpoints may be reduced if  $p_3$  knows sizes of the respective log information for  $p_1, p_2$  and  $p_4$  in its volatile storage. CCPNA obtains such information by maintaining an array,  $LogSize_p$ , to save the size of the log information in the volatile storage by process. Thus, CCPNA can reduce the number of additional messages and forced checkpoints by using the vector compared with the traditional algorithm.

- $log_p$ : It is a set saving  $e(rid, ssn, rsn, data)$  of each message sent by  $p$ . It is initialized to  $\emptyset$ .
- $Lssn_p$ : It is the send sequence number of the latest message sent by  $p$ . It is initialized to 0.
- $Lrsn_p$ : It is the receive sequence number of the latest message delivered to  $p$ . It is initialized to 0.
- $LssnVec_p$ : It is a vector where  $LssnVec_p[q]$  records the send sequence number of the latest message received by  $p$  that  $q$  sent. Each element of the vector is initialized to 0.
- $LogSize_p$ : It is a vector where  $LogSize_p[q]$  is the sum of sizes of all  $e(m)$ s in  $log_p$  such that  $p$  sent  $m$  to  $q$ .  $LogSize_p[q]$  is initialized to 0.
- $LrsnInLchkpt_p$ : It is the  $rsn$  of the latest message delivered to  $p$  before  $p$ 's having taken its last checkpoint. It is initialized to 0.
- $ENsend_p$ : It is a set of  $rsns$  that aren't yet recorded at the senders of their messages. It is initialized to an empty set  $\Phi$ . It is used for indicating whether  $p$  can send messages to other processes (when  $ENsend_p = \Phi$ ) or not.

**Fig. 2.** Data Structures for every process  $p$  in CCPNA



**Fig. 3.** An example of executing our algorithm CCPNA

In CCPNA, each process  $p$  should maintain the data structures shown in figure 2. First,  $LogSize_p$  is a vector where  $LogSize_p[q]$  is the sum of sizes of all  $e(m)$ s in  $log_p$ , 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 more empty buffer space, it executes CCPNA. 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  $\exists e(m) \in log_p$ . When

---

```

procedure MSEND(data, q)
  wait until ( $ENsend_p = \Phi$ ) ;
   $Lssn_p \leftarrow Lssn_p + 1$  ;
  send  $m(Lssn_p, data)$  to q ;
   $log_p \leftarrow log_p \cup \{(q, Lssn_p, -1, data)\}$  ;
   $LogSize_p[q] \leftarrow LogSize_p[q] + \text{size of } (q, Lssn_p, -1, data)$  ;

procedure MRECV( $m(ssn, data)$ , sid)
  if ( $LssnVec_p[sid] < m.ssn$ ) then {
     $Lrsn_p \leftarrow Lrsn_p + 1$  ;
     $LssnVec_p[sid] = m.ssn$  ;
    send  $ack(m.ssn, Lrsn_p)$  to sid ;
     $ENsend_p \leftarrow ENsend_p \cup \{Lrsn_p\}$  ;
    deliver  $m.data$  to the application ;
  }
  else discard  $m$  ;

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

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

procedure CHECKPOINTING()
   $LrsnInLchkpt_p \leftarrow Lrsn_p$  ;
  take its local checkpoint on the stable storage ;

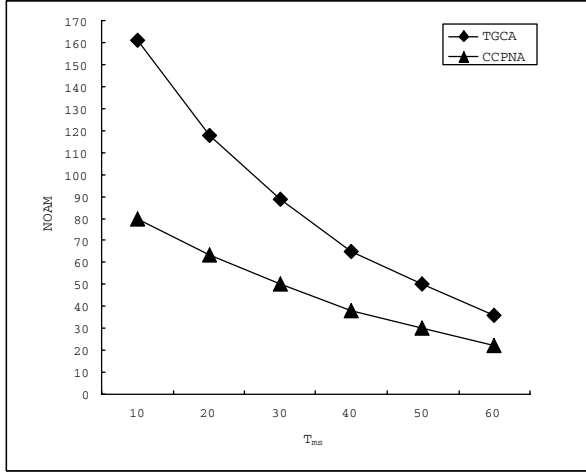
procedure AGC( $sizeOflogSpace$ )
   $participatingProcs \leftarrow \emptyset$  ;
  while  $sizeOflogSpace > 0$  do
    if (there is  $r$  st  $((r \in P) \wedge (r \text{ is not an element of } participatingProcs) \wedge$ 
       $(LogSize_p[r] \neq 0) \wedge (\max LogSize_p[r]))$ ) then {
       $sizeOflogSpace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

procedure CHECKLRSNINLCHKPT( $Request(MaximumRsn), q$ )
  if ( $LrsnInLchkpt_p < MaximumRsn$ ) then
    CHECKPOINTING() ;
  send  $Reply()$  to  $q$  ;

```

---

Fig. 4. Procedures for every process  $p$  in CCPNA



**Fig. 5.** *NOAM* vs.  $T_{ms}$

$q$  receives the request message with the  $rsn$  from  $p$ , it checks whether the  $rsn$  is greater than  $LrsnInLchkpt_p$ . If so, it should take a checkpoint and then send  $p$  a reply message. Otherwise, it has only to send  $p$  a reply message. When  $p$  receives the reply message from  $q$ , it removes all  $e(m)$ s from  $log_p$  such that the receiver of  $m$  is  $q$ .

For example, in figure 3, when  $p_3$  attempts to execute CCPNA at the marked point after it has sent  $m_8$  to  $p_2$ , it should create *participatingProcs*. In this figure, we can see that  $LogSize_{p_3}[p_2](= Y)$  is the largest ( $Y \geq Z \geq X$ ) among all the elements of  $LogSize_{p_3}$  due to  $e(m_1)$ ,  $e(m_2)$ ,  $e(m_4)$ ,  $e(m_6)$  and  $e(m_8)$  in  $log_{p_3}$ . Thus, it first selects and appends  $p_2$  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  $p_1$  and  $p_4$  no longer. Therefore,  $p_3$  sends a checkpoint request message with  $m_8.rsn$  to only  $p_2$  in *participatingProcs*. When  $p_2$  receives the request message, it should take a forced checkpoint like in this figure because the  $rsn$  included in the message is greater than  $LrsnInLchkpt_{p_2}$ . Then it sends  $p_3$  a reply. When  $p_3$  receives a reply message from  $p_2$ , it can remove  $e(m_1)$ ,  $e(m_2)$ ,  $e(m_4)$ ,  $e(m_6)$  and  $e(m_8)$  from  $log_{p_3}$ . From this example, we can see that CCPNA chooses a small number of processes to participate in the garbage collection based on  $LogSize_{p_3}$  compared with the traditional algorithm. Thus, CCPNA may reduce the number of additional messages and forced checkpoints.

### 3.1 Algorithmic Description

The procedures for process  $p$  in our algorithm are formally described in figure 4.  $MSEND()$  is the procedure executed when each process  $p$  sends a message  $m$  and logs the message to its volatile memory. In this case,  $p$  adds the size of  $e(m)$  to  $LogSize_p[q]$  after transmitting the message. Procedure  $MRECV()$

is executed when  $p$  receives a message. 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,  $LrsnInLchkpt_p$  is updated to the  $rsn$  of the last message received before the checkpoint. `AGC()` is the procedure executed when each process attempts to initiate the forced garbage collection, and `CHECKLRSNINLCHKPT()` is the procedure for forcing the log information to become useless for future recovery.

## 4 Performance Evaluation

In this section, we perform extensive simulations to compare the proposed algorithm CCPNA with the traditional algorithm TGCA using simjava discrete-event simulation language [7]. Two performance indexes are used for comparison; the average number of additional messages (*NOAM*) and the average number of forced checkpoints (*NOFC*) required for garbage collection per process. In the literature, these two indexes dominate the overhead caused by garbage collection during failure-free operation [5]. A system with 20 nodes connected through a general network was simulated. Each node has one process executing on it and, for simplicity, the processes are assumed to be initiated and completed together. The message transmission capacity of a link in the network is 100Mbps. For the simulation, 20 processes have been executed for 72 hours per simulation run. Every process has a 10MB buffer space for storing its  $log_p$ . The message size ranges from 50KB to 200KB. Normal checkpointing is initiated at each process with an interval following an exponential distribution with a mean  $T_{ckpt}=360$  seconds. The simulation parameter is the mean message sending interval,  $T_{ms}$ , following an exponential distribution.

Figure 5 shows *NOAM* for the various  $T_{ms}$  values. In this figure, we can see that *NOAMs* of the two algorithms increase as  $T_{ms}$  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 CCPNA is much lower than that of TGCA. CCPNA reduces about 38% - 50% of *NOAM* compared with TGCA.

Figure 6 illustrates *NOFC* for the various  $T_{ms}$  values. In this figure, we can also see that *NOFCs* of the two algorithms increase as  $T_{ms}$  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 CCPNA is lower than that of TGCA. CCPNA reduces about 25% - 51% of *NOFC* compared with TGCA.

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



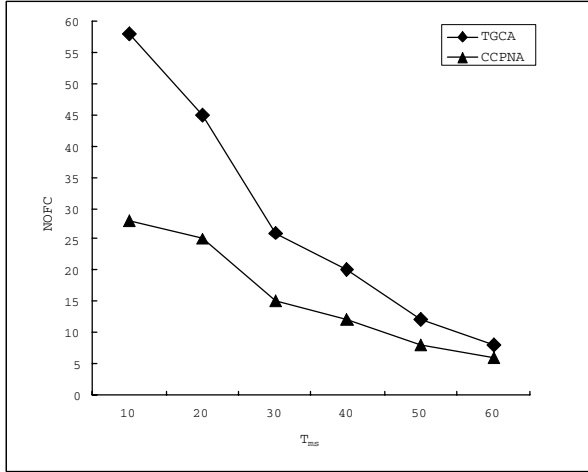


Fig. 6. *NOFC* vs.  $T_{ms}$

## 5 Conclusion

In this paper, we presented a garbage collection algorithm CCPNA for efficiently removing log information of each process in sender-based message logging. CCPNA allows each process to keep an array to save the size of the log information for every process in its storage by process. It chooses a minimum number of processes to participate in the forced garbage collection based on the array. Thus, it incurs more additional messages and forced checkpoints than our previous algorithm. However, it can avoid the risk of overloading the storage buffers unlike the latter. Moreover, CCPNA reduces the number of additional messages and forced checkpoints needed by the garbage collection compared with the traditional algorithm TGCA. From our simulation experiments, we can see that CCPNA significantly reduces about 38% - 50% of NOAM and 25% - 51% of NOFC regardless of the communication patterns compared with TGCA.

## References

1. JinHo Ahn. An Efficient Algorithm for Removing Useless Logged Messages in SBML Protocols. *Lecture Notes In Computer Science*, Vol. 3816, pp. 166-171, Dec. 2005.
2. 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.
3. K. M. Chandy, and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1): 63-75, 1985.

4. 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.
5. 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.
6. L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21, pp. 558-565, 1978.
7. 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.
8. 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.
9. P. Sens and B. Folliot. The STAR Fault Tolerant manager for Distributed Operating Environments. *Software Practice and Experience*, 28(10), pp. 1079-1099, 1998
10. 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.
11. R.E. Strom, D.F. Bacon and S.A. Yemeni. Volatile Logging in n-Fault-Tolerant Distributed Systems. *In Digest of Papers: the 18th International Symposium on Fault-Tolerant Computing*, pp. 44-49, 1988.
12. R.E. Strom and S.A. Yemeni. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3, pp. 204-226, 1985.
13. 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.
14. 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.