International Journal of Grid and Distributed Computing
Vol. 11, No. 4 (2018), pp.13-26
http://dx.doi.org/10.14257/ijgdc.2018.11.4.02

Efficient Mechanism for Enhancing Recovery Performance of SBML in Hierarchical and Distributed Server Architectures

Jinho Ahn*

Dept. of Computer Science, Kyonggi University, Suwon-si, Gyeonggi-do, Korea
Tel.: +82 31 249-9674, FAX: +82 31 249-9673
jhahn@kgu.ac.kr

Abstract

The application of the original sender-based message logging(SBML) to the hierarchical architecture may degrade scalability in case of node failure occurrences due to its behavioral limitation. To address this issue, area leader-based mechanism(ALBM) was proposed, but forces all loads of logging and maintaining inter-area messages destined to an area to become concentrated on its leader. In this paper, we propose an efficient mechanism to keep the contents and the receive sequence number of each inter-area transmitted message into the area-wide stable storage of its receiver's area. The mechanism allows each recovering node to locally restore to be in a consistent state without any help of the other areas. Also, it allows the logging procedure of each inter-area message to be achieved at its own receiver, not the area leader. The experimental results illustrate our mechanism is superior to the previous one, ALBM, in terms of message logging and log information maintenance load.

Keywords: *Hierarchical and Distributed Architecture, Fault-tolerance, Message Logging and Checkpointing, Recovery*

1. Introduction

Recently, commonly used computing environments tend to have a variety of computational services that should perform big jobs [11, 17, 25]. In order to reduce response time of each service in these environments, they generally make the job of each service divided into a set of tasks and them evenly distributed and executed onto multiple nodes in a server area in parallel. Especially, as complexity of such a computational service has greatly increased, the flat-style parallel computing architecture consisting of a set of nodes in an area reaches its inherent limit of scalability [4, 10]. This desire for higher performance makes many organizations adopt the hierarchical architecture. The architecture is generally composed of a group of node areas that possess a certain number of computational nodes respectively.

These distributed computing architectures require efficient fault-tolerance techniques for improving service availability in case of sequential node failures [5, 7, 9, 18, 22]. For this purpose, checkpointing and message logging may satisfy the requirement with much less computing resources during failure-free operation than process replication [12]. Among the message logging algorithms, Sender-Based Message Logging(SBML) [2, 6, 13-15, 19-20, 26] can greatly reduce the cost of synchronous logging to the stable storage compared with receiver-based pessimistic message logging [21, 27] by maintaining the log information of each message onto the volatile memory buffer of its sender. Also, if nodes or processes sequentially fail, they can obtain the contents and the receive sequence number(rsn) of each message from its sender, and replay it in a pre-failure order,

Received (October 5, 2017), Review Result (January 10, 2018), Accepted (January 16, 2018)
* Corresponding Author

ISSN: 2005-4262 IJGDC

recovering to reach their consistent states. Thanks to these desirable properties, SBML has been frequently used in flat-style architectures.

However, when the original SBML [6, 13-15, 19-20, 26] is applied into the hierarchical server architecture, the following structural feature should certainly be considered. Generally, when a big job is executed on the hierarchical server architecture, the job should be divided into a set of tasks to be able to provide its final result for service clients within shorter response time. At this point, it should be essentially considered that the inter-area communication cost is much higher than the intra-area one. Therefore, in order to complete execution of the corresponding job as soon as possible, it should be classified into a certain number of subsets of tightly-coupled tasks according to the number of available areas and the number of available nodes per area. Then, each subset of tasks is dispatched to an appropriate area. Due to this property, the original SBML incurs the following drawback that it may degrade scalability in case of node failure occurrences.

- It is possible that a node received inter-area communication messages from an arbitrary number of distinct areas before its failure. When the node crashes, it may have to ask every area and all its nodes the recovery information for it during the initial step of the recovery procedure, which leads to highly lengthening its recovery time because of very high communication latency.

In order to address this problem, the recovery information of each message should be safely preserved in the area that its receiver belongs to. One research work [2] has been performed to attempt to address this issue. In the system assumed, a leader is elected among a group of processes or nodes, and every message from another group is hierarchically transmitted to its receiver via the leader. The mechanism proposed in the previous work forces the leader to be the virtual sender of the message and keep all the log information of the message in its volatile storage. Thanks to this feature, if several processes in a group except the leader crash, they can restore to its pre-failure state by obtaining the log information of each message received before failure from the leader without any help of the real message senders in other areas. However, the leader should solely be responsible for logging every inter-area message received and maintaining its log information in its own memory buffer. This behavioral feature may significantly degrade scalability of the system.

This paper presents an efficient mechanism to keep the contents and the rsn of each inter-area transmitted message into the area-wide stable storage of its receiver's area. The mechanism allows each recovering node to locally restore to be in a consistent state without any help of the other areas. This positive feature results in high reduction of the recovery time of every inter-area communication message received before its receiver's failure compared with the previous SBML. It may be an essential property to ensure required quality of service and reasonable response time to clients.

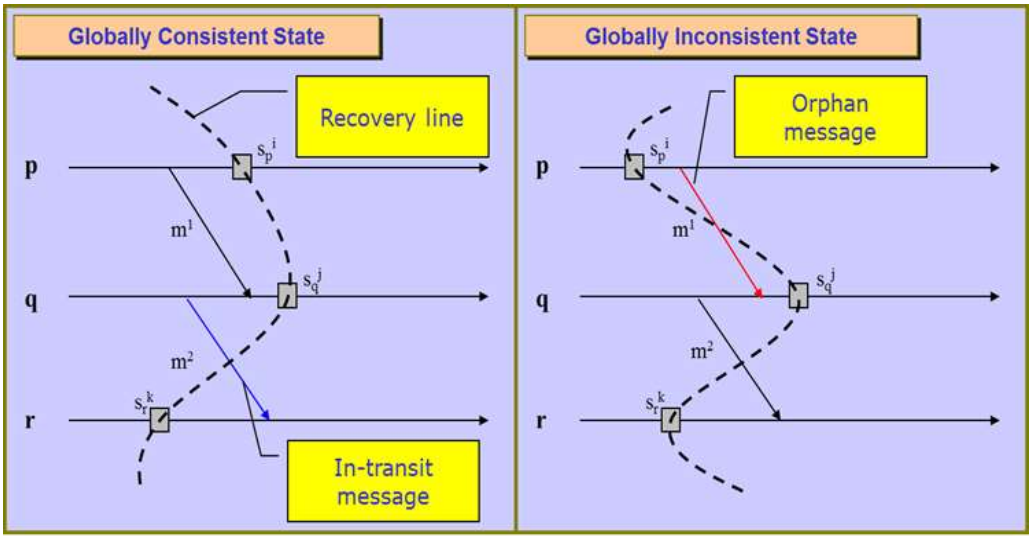
The remainder of the paper is structured as follows. In Section 2, we describe the system model assumed and, in Section 3, the limitation of the previous SBML in detail. Section 4 describes our mechanism and Section 5 evaluates the mechanism over the original one. In Section 6, we conclude this paper.

2. System Model

A distributed computation consists of a set P of $n(n > 0)$ sequential processes executed on sensor nodes 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 [12]. Processes have no global memory and global clock. The system is asynchronous: each process is executed at its own speed and communicates with each other only through messages at finite but arbitrary

transmission delays. Exchanging messages may temporarily be lost but, eventually delivered in FIFO order. We assume that the communication network is immune to partitioning and nodes fail according to the fail stop model where every crashed process on them halts its computation with losing all contents of its volatile memory [23]. 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 [15, 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 $si^k_p (k > 0)$, is started by the delivery event of the k-th message m of p, denoted by $dev^k_p(m)$. Therefore, given p's initial state, si^0_p, and the non-deterministic events, $[dev^1_p, dev^2_p, ..., dev^i_p]$, its corresponding state s^i_p is uniquely determined. Let p's state, $s^i_p = [si^0_p, si^1_p, ..., si^i_p]$, represent the sequence of all state intervals up to si^i_p. s^i_p and $s^j_q (p \neq q)$ are mutually consistent if all messages from q that p has delivered to the application in s^i_p were sent to p by q in s^j_q, and vice versa [8]. 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.

To understand these definitions precisely, figure 1 shows two examples of global states, which are shown by broken arrows. In figure 1(a), states s^i_p and s^j_q are mutually consistent because they reflect sending and receiving message m^1 respectively. Message m^2 has been sent in state s^j_q but not yet received in state s^k_r. The states s^j_q and s^k_r are also mutually consistent because the situation where the message m^2 has been in transit could have occurred in a failure-free and correct execution. We call such a message an in-transit message. Therefore, the global state in this figure, consisting of s^i_p, s^j_q and s^k_r, is consistent. However, in figure 1(b), states s^i_p and s^j_q are mutually inconsistent because though message m^1 has not been left in the state s^i_p, the state s^j_q has reflected receiving the message. Such a message like m^1 is named orphan message. Here, orphan message means the message received from a process though there is no record that it was sent from the process due to process failures. Message m^1 may make the state of q, s^j_q, inconsistent with those of the other live processes after recovery. At this time, the receiver of m^1, q, is called orphan process. Thus, the states, s^i_p, s^j_q and s^k_r, in this figure compose a globally inconsistent state.



(a) An example of a globally consistent state (b) An example of a globally inconsistent state

Figure 1. Examples Illustrating how to Decide Whether the State is Globally 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. Previous SBML Mechanisms

Originally, sender-based message logging is designed to have the positive feature of receiver-based pessimistic message logging, no roll-back property, in case of sequential failures. Also, it may significantly reduce high failure-free overhead resulting from the disadvantageous feature of the latter, *i.e.*, synchronous logging on stable storage as soon as each message is received or before any message generated after the received message is sent to another process. To satisfy these requirements, this technique allows each received message to be logged on the volatile storage of its sender, called semi-synchronous logging. Also, to ensure system consistency in case a process crashes at a time, the log information of each message received by the process is forced to save into its sender's volatile storage before sending another process any message generated after the receipt of the former message.

Let us closely examine how sender-based message logging can have the desirable feature mentioned above using Figure 2. In the figure, there are four processes, p, q, r, and s. Here, p sends several messages to q, r, and s respectively. In this operation, the sender p records the partial log information of the corresponding sent message on its own volatile memory. At this point, the log information of a message m is composed of four elements, the send sequence number (ssn), the receive sequence number (rsn), the receiver's id (rid) and data of the message. Here, partially logged(smi(m)) means the rsn of the message has not been recorded on the log information yet. Then, each receiver q, r, and s first receives message m from p, increments its rsn, rsn_m, by one, and assigns the value of rsn_m to the message. Next, the receiver sends sender p an acknowledgment message including rsn_m. At this point, it keeps a determinant of m, det(m), on its memory buffer for giving rsn_m to m's sender in case of the latter's failure. Here, det(m) consists of the identifier of the sender(sid), ssn, rid, and rsn. When sender p obtains an acknowledgment message for its sent message m from the corresponding receiver, it updates m's log information with rsn_m attached to the acknowledgment. At this time, m is called fully logged(smi(m) with rsn_m), meaning every element in the log information of m is filled with its actual value for m's recovery. Then, the sender notifies the corresponding receiver of the fact that it safely holds the full log information on its own volatile memory.

However, if a node on the hierarchical architecture crashes, the original SBML [6, 13-15, 19-20, 26] forces the node to gather the contents and the rsn of every inter-area message received before its failure in the initial phase of the recovery procedure. In this situation, the total recovery latency, resulting from the phase, may be very high. In the worst case, the failed node should transmit each recovery request message to and get the recovery information for the node from every node in all the areas. Let us examine the shortcoming of the original SBML using an example. Figure 3 shows its recovery procedure in case the first process in area 2, denoted by $P_{(2.1)}$, fails. Before failure, $P_{(2.1)}$ received two messages m_4 and m_6 from $P_{(1,4)}$ in area 1 and $P_{(3,3)}$ in area 3. Therefore, $P_{(2.1)}$ sends each recovery request message, $request_{2.1}$, to the two message senders. Receiving $request_{2.1}$, they transmit the contents and the rsns of m_4 and m_6 to $P_{(2.1)}$ like in this figure. Afterwards, $P_{(2.1)}$ can replay the two messages in a consistent order with their log information. From this figure, we can recognize that, in the original SBML, the number of inter-area control messages generated during the recovery procedure linearly grows together with the number of areas and the number of nodes belonging to each area in the system.

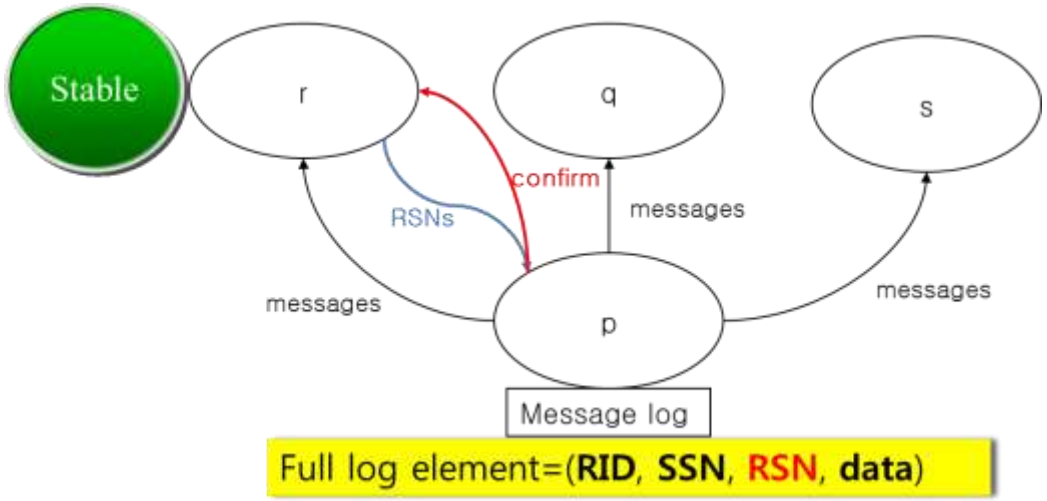


Figure 2. Illustration Showing the Procedure of the Original SBML

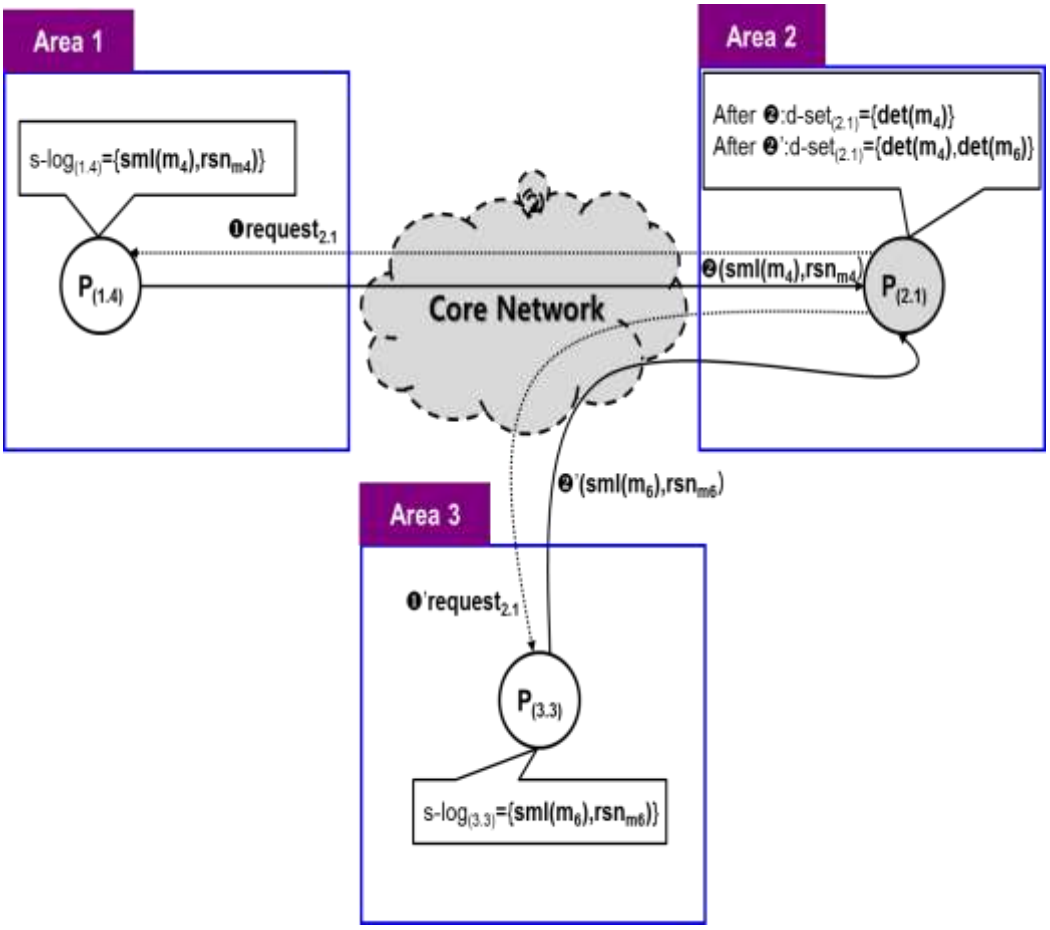


Figure 3. Example Showing the Drawback of the Original SBML

Area leader-based SBML [2] has been proposed to address this problem. The mechanism has all inter-area messages from other areas destined to an area first transmitted to its leader. For example, in figure 4, message m_4 $P_{(1,4)}$ sends to $P_{(2.1)}$ is first forwarded to the leader of area 2. Afterwards, the leader plays the role of the sender of m_4 for $P_{(2.1)}$ in area 2, called virtual sender. Thus, the leader maintains the contents and the rsn of m_4 in its volatile memory buffer. If $P_{(2.1)}$ crashes, it can complete its recovery

procedure by collecting the log information of every inter-area message like m_4 it received before failure from the leader. However, the mechanism forces all loads of logging and maintaining inter-area messages destined to an area to become concentrated on its leader, which may highly degrade scalability during failure-free operation. Especially, if the leader fails, it should collect not only the log information for itself, but also the contents of every inter-area message destined to another process in the same area.

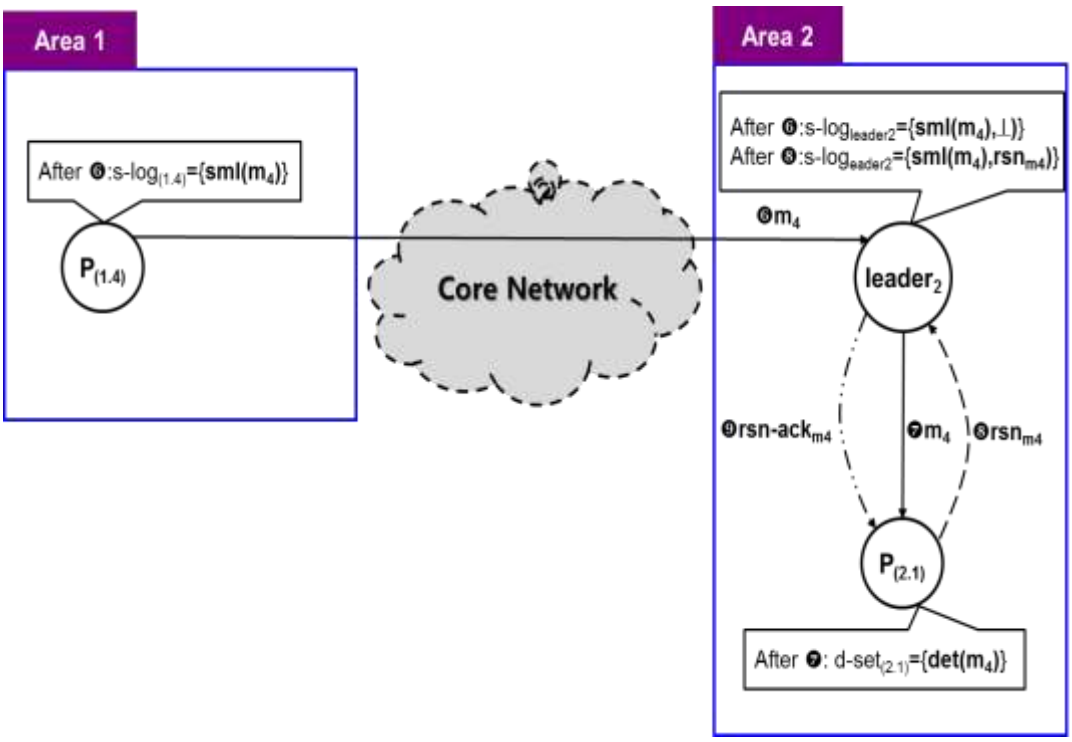


Figure 4. Example Showing the Logging Procedure of the Area Leader-Based SBML [2]

4. The Proposed SBML Mechanism

Recently, the hierarchical architecture is preferred to the flat-style one to improve scalability. The structural feature may make the application of the original SBML to the first not suitable due to its inadequacy that it is designed appropriate for the second. In order to address this drawback, we attempt to design a simple and efficient mechanism having the following desirable features.

- Enable complete recovery without any help of other nodes in different areas in case of sequential failures.
- Allow each node, not a centralized component like area leader, to be responsible of logging every inter-area message destined to it.

To have these features, the mechanism uses the area-wide stable storage for saving each inter-area message destined to a node in its area. With this behavioral feature, the log information of intra-area transmitted messages is maintained by their respective senders while each node receiving messages from other areas is responsible for saving their log information onto the area-wide stable storage for localized recovery.

Let us show how our mechanism can satisfy these requirements with several examples. When any message from an area is transmitted to another area (by executing Module **SEND_MSG**(aid, pid, data) AT $P_{(i.j)}$), its receiver executes the module for processing

inter-area messages (by executing Module **RECEIVE_MSG**(m(aid, pid, ssn, data)) AT $P_{(i.j)}$). Figure 5 shows an example of this case that $P_{(1.4)}$ sends message m_4 to $P_{(2.1)}$. In this example, $P_{(1.4)}$ need not save smi(m_4) in its buffer s-$log_{(1.4)}$. $P_{(2.1)}$ first saves smi(m_4) and rsn_{m4} in the area-wide stable storage belonging to area 2. Afterwards, as $P_{(2.1)}$ sends m_5 to another member of the same area, $P_{(2.2)}$, it keeps smi(m_5) in its buffer s-$log_{(2.1)}$ without rsn_{m5}. When $P_{(2.2)}$ receives m_5, it assigns rsn_{m5} to the message and then saves det(m_5) in its buffer d-$set_{(2.2)}$ before sending it to $P_{(2.1)}$. Receiving rsn_{m5}, $P_{(2.1)}$ can update the element for m_5 in s-$log_{(2.1)}$ with it and then confirms $P_{(2.2)}$ the receipt of rsn_{m5} (by executing Module **RECEIVE_RSN**(rsn-return(aid, pid, ssn, rsn)) AT $P_{(2.1)}$ and Module **RECEIVE_RSN_ACK**(rsn-ack(aid, pid, rsn)) AT $P_{(2.2)}$).

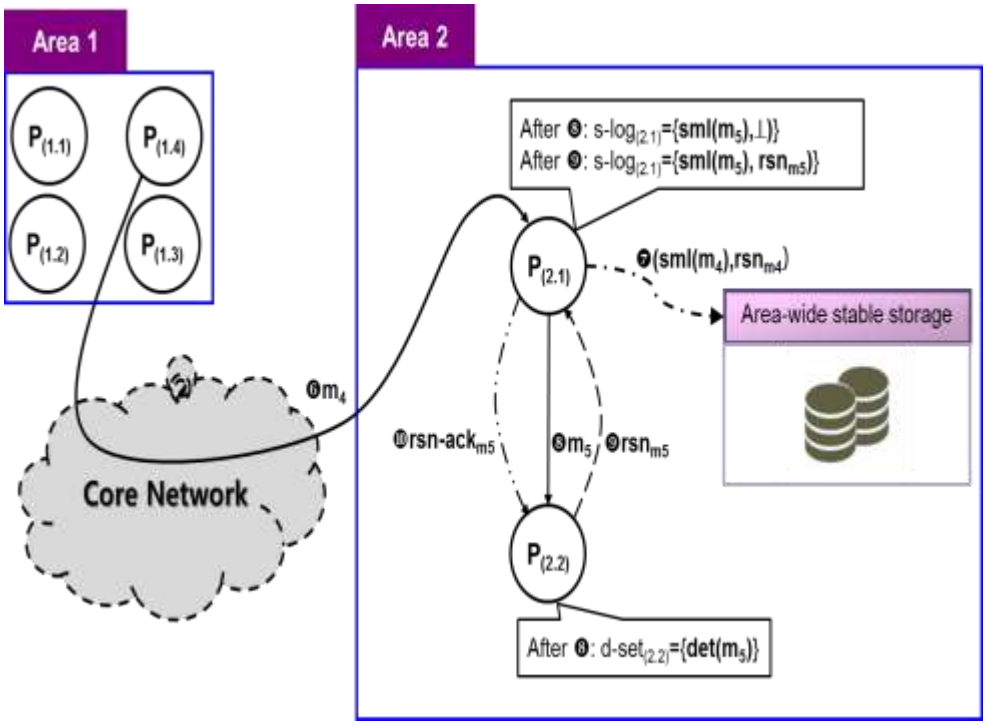


Figure 5. Example Showing how our Mechanism Handles both Inter-Area and Intra-Area Transmitted Messages

If $P_{(2.1)}$ crashes like in Figure 6, it can get m_4 and its rsn from its local area-wide stable storage without any help of the sender of m_4, $P_{(1.4)}$ (by executing Module **RECOVERY**() AT $P_{(2.1)}$). Also, $P_{(2.1)}$ regains rsn_{m5} from $P_{(2.2)}$ (by executing Module **RECEIVE_MSG**(m(aid, pid, ssn, data)) AT $P_{(2.2)}$) and can complete the log information for m_5 in its buffer s-$log_{(2.1)}$. Secondly, like in Figure 7, when a node in the same area of $P_{(2.1)}$, $P_{(2.2)}$, fails, $P_{(2.1)}$ can directly provide smi(m_5) and rsn_{m5} for $P_{(2.2)}$ (by executing Module **RECEIVE_ REQUEST**(request(pid)) AT $P_{(2.1)}$). From these examples, we can see that our mechanism enables the logging procedure of each inter-area message to be achieved at its own receiver, not the area leader, improving scalability in terms of message logging and log information maintenance load. Also, the mechanism can fulfill area-wide recovery even if there are a certain number of inter-area messages transmitted to the corresponding area.

The algorithmic description of message logging and recovery procedures of our mechanism is shown in Figures 8 and 9.

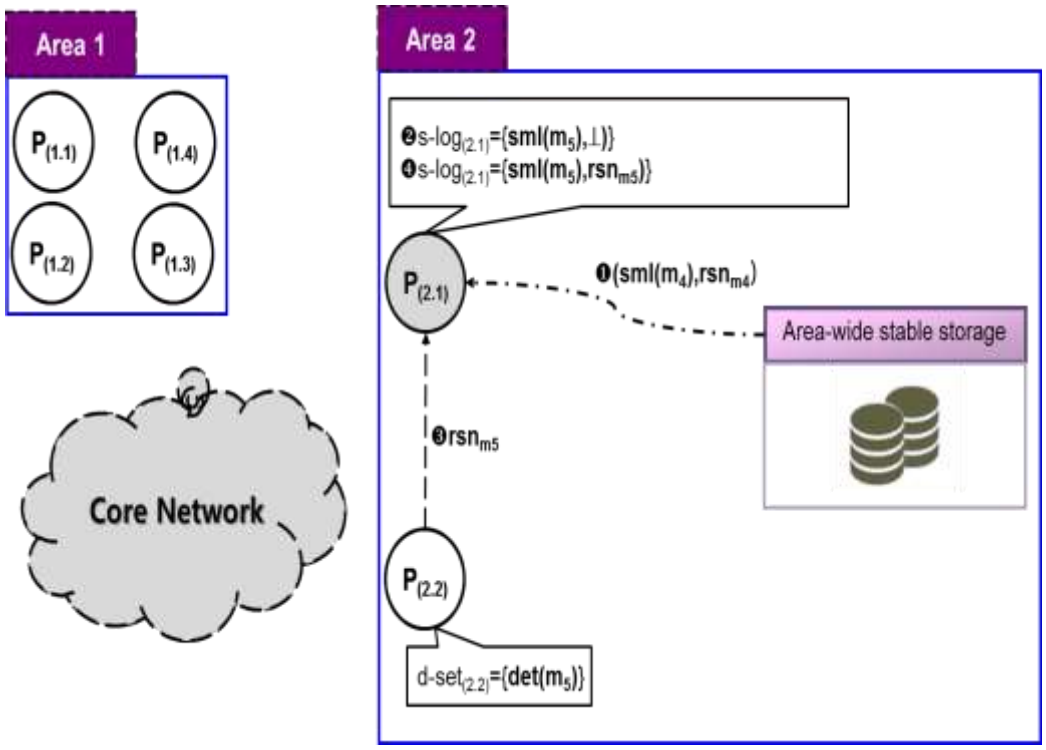


Figure 6. Example Showing how to Recover Inter-Area Messages in our Mechanism

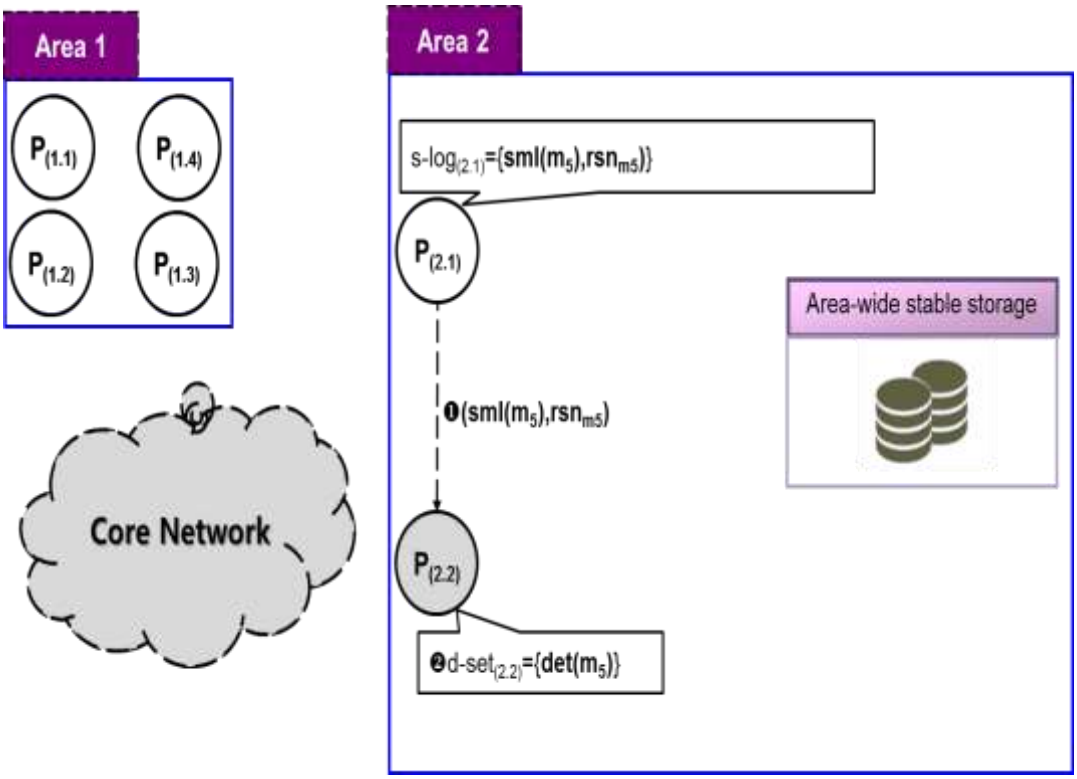


Figure 7. Example Showing how to Recover Intra-Area Messages in our Mechanism

```
Module SEND_MSG(aid, pid, data) AT P(i.j)
   SSN(i.j) ← SSN(i.j)+1;
   send m(SSN(i.j), data) to P(aid.pid);
   if(aid = i) then
      s-log(i.j) ← s-log(i.j) ∪ {(aid, pid, SSN(i.j), data, ⊥)};

Module RECEIVE_MSG(m(aid, pid, ssn, data)) AT P(i.j)
   if(SSNTable(i.j)[m.aid, m.pid] < m.ssn) then
     RSN(i.j) ← RSN(i.j)+1;
     SSNTable(i.j)[m.aid, m.pid] ← m.ssn;
     if(aid = i) then
       send rsn-return(i, j, m.ssn, RSN(i.j)) to P(m.aid.m.pid);
       d-set(i.j) ← d-set(i.j) ∪ {(m.aid, m.pid, m.ssn, RSN(i.j))};
       delay all the send message operations generated after having received m;
     else
       save (m.aid, m.pid, m.ssn, RSN(i.j), m.data) into flog(j) in area-wide stable
          storage(i);
     deliver m.data to its corresponding application;
   else
     find ∃e ∈ d-set(i.j) st ((m.aid = i) ^ (m.pid = e.pid) ^ (m.ssn = e.ssn));
     send rsn-return(i, j, m.ssn, e.rsn) to P(m.aid.m.pid);

Module RECEIVE_RSN(rsn-return(aid, pid, ssn, rsn)) AT P(i.j)
   find ∃e ∈ s-log(i.j) st ((rsn-return.aid = e.aid) ^ (rsn-return.pid = e.pid)
      ^ (rsn-return.ssn = e.ssn));
   e.rsn ← rsn-return.rsn;
   send rsn-ack(i, j, rsn-return.rsn) to P(rsn-return.aid. rsn-return.pid);

Module RECEIVE_RSN_ACK(rsn-ack(aid, pid, rsn)) AT P(i.j)
   if(stableRSN(i.j) < rsn-ack.rsn) then
      allow all the send message operations delayed before receiving the message
whose
         RSN value is (rsn-ack.rsn+1) to begin executing;
      stableRSN(i.j) ← rsn-ack.rsn;

Module TAKE-CHECKPOINT() AT P(i.j)
   take its local checkpoint with (RSN(i.j), SSN(i.j), SSNTable(i.j), s-log(i.j))
      on area-wide stable storage(i);
   allow all the send message operations delayed before this checkpoint
      to begin executing;
   stableRSN(i.j) ← RSN(i.j);
```

Figure 8. Message Logging and Checkpointing Procedures

```
Module RECOVERY() AT P(i.j)
  restore a latest checkpointed state with (RSN(i.j), SSN(i.j), SSNTable(i.j), s-log(i.j), flog(j))
    from area-wide stable storage(i);
  broadcast each a recovery request message, request(pid), to all the others in area i;
  fullL ← fullL ∪ flog(j);
  while there remain any recovery replies needed for making P(i.j) consistent do
    receive a reply r from P(i.k);
    put fully logged messages for P(i.j) piggybacked on r into fullL in RSN order;
    put partially logged messages for P(i.j) piggybacked on r into partialL in FIFO order;
  RSN(i.j) ← RSN(i.j)+1;
  for ∀e ∈ fullL st (e.rsn = RSN(i.j)) do
    deliver e.data to its corresponding application;
    SSNTable(i.j)[e.aid, e.pid] ← e.ssn;
    if(aid = i) then
      d-set(i.j) ← d-set(i.j) ∪ {(e.aid, e.pid, e.ssn, RSN(i.j))};
    RSN(i.j) ← RSN(i.j)+1;
    remove e from fullL;
  stableRSN(i.j) ← RSN(i.j);
  while partialL is a non-empty set do
    select ∃ e ∈ partialL in FIFO order ;
    call Module RECEIVE_MSG(e.aid, e.pid, e.ssn, e.data) at P(i.j);
    remove e from partialL;

Module RECEIVE_ REQUEST(request(pid)) AT P(i.j)
  put fully and partially logged messages for request.pid in s-log(i.j) into a reply r;
  send r to P(i.request.pid);
```

Figure 9. Recovery Procedures

5. Performance Evaluation

In this section, we have performed extensive simulations to evaluate performance of the two mechanisms, ALBM(Area Leader-Based Mechanism) [2] and our mechanism, OURS, using a discrete-event simulation language named PARSEC [3]. Whenever a node in an area is the target of an inter-area message m from another area, ALBM has the area leader always play the role of sender of the message as virtual sender by recording and maintaining its log information onto its memory buffer. With this feature, the mechanism results in no additional inter-area control message the original SBML [6, 13-15, 19-20, 26] requires. Also, OURS needs no extra inter-area control message because it can locally complete recovery using the area-wide stable storage. Therefore, we use one performance index for comparison of failure-free overhead to consider as follows; the elapsed time until the same distributed execution has been completed ($T_{complete}$). A simulated system is composed of M areas each associated with a coordinate(x_{area}, y_{area}). Any two adjacent areas are connected with a link having a bandwidth of 1 Gbps. Hierarchically, an area is composed of N nodes each associated with a coordinate(x_{node}, y_{node}). Any two adjacent nodes are connected with a link having a bandwidth of 100 Mbps.

For simplicity of this simulation, we assume that both bandwidth and propagation delay between any pair of nodes are proportional to their distance. Each node executes one process, and for simplicity, it is assumed that the processes are initiated and completed simultaneously. The target of each application message sent from a process is always one process. Every process has a 128MB buffer space for storing its message log. The size of application messages ranges from 1KB to 100KB. Normal checkpointing is performed at each process periodically with the interval T_{nc}, following an exponential distribution with a mean value of 300 seconds. In addition, a message to a process is sent with an interval T_{ms}, following an exponential distribution with a mean value of T_{ms}=100ms. All experimental results shown in this simulation are all averages over a number of trials. Distributed applications used for the simulation exhibit the following four communication patterns, respectively [1].

• Serial pattern: All process groups are organized in a serial manner and transfer messages for one way. When a process group, except the first and the last ones, receives a message from its predecessor, it sends a message to its successor, and vice versa. The first process group communicates with only its successor and the last one communicates with its predecessor only.
• Circular pattern: A logical ring is structured for communication among process groups in this pattern. Every process group communicates with only two directly connected neighbors.
• Hierarchical pattern: A logical tree is structured for communication among process groups in this pattern. Every process group, except one root group, communicates with only one parent process group and k child process groups ($k \geq 0$). The root group communicates with its child group only.
• Irregular pattern: The communication among process groups follows no special communication pattern. Here, a message to a process group is sent from a randomly chosen process group.

Figure 10 shows $T_{complete}$ for the two mechanisms, ALBM and OURS, for four different communication patterns with varying the number of nodes per area, $area_{size}$, ranging from 5 to 20 in case the number of areas is 5. As $area_{size}$ becomes bigger, $T_{complete}$ also decreases in all four patterns because the total number of nodes in the system is larger accordingly. However, these subfigures indicate that, regardless of the application communication patterns, when $T_{complete}$ of OURS is much smaller than that of ALBM. The first reduces approximately 17.9%~38.1% of $T_{complete}$ compared with the second. This phenomenon results from the reason that OURS can distribute the loads of saving and keeping every inter-area transmitted message to its own receiver unlike ALBM. From this outcome, we can see OURS enhances scalability over ALBM in terms of message logging and log information maintenance load.

6. Conclusion

In this paper, a simple and efficient SBML mechanism has been designed with the two desirable features. First, the mechanism enables complete recovery without any help of other nodes in different areas in case of sequential failures. Secondly, it allows the logging procedure of each inter-area message to be achieved at its own receiver, not the area leader. It can achieve these features by using the area-wide stable storage for saving each inter-area message destined to a node in its area. The proposed mechanism may improve scalability in terms of message logging and log information maintenance load. Also, the mechanism can swiftly fulfill area-wide recovery even if there are a certain number of inter-area messages transmitted to the corresponding area. The experimental results illustrate that our mechanism can be a fast recovery enabling technique with little

normal operation overhead for lifting the behavioral limitation of the original SBML in case of its application to hierarchical and distributed architectures.

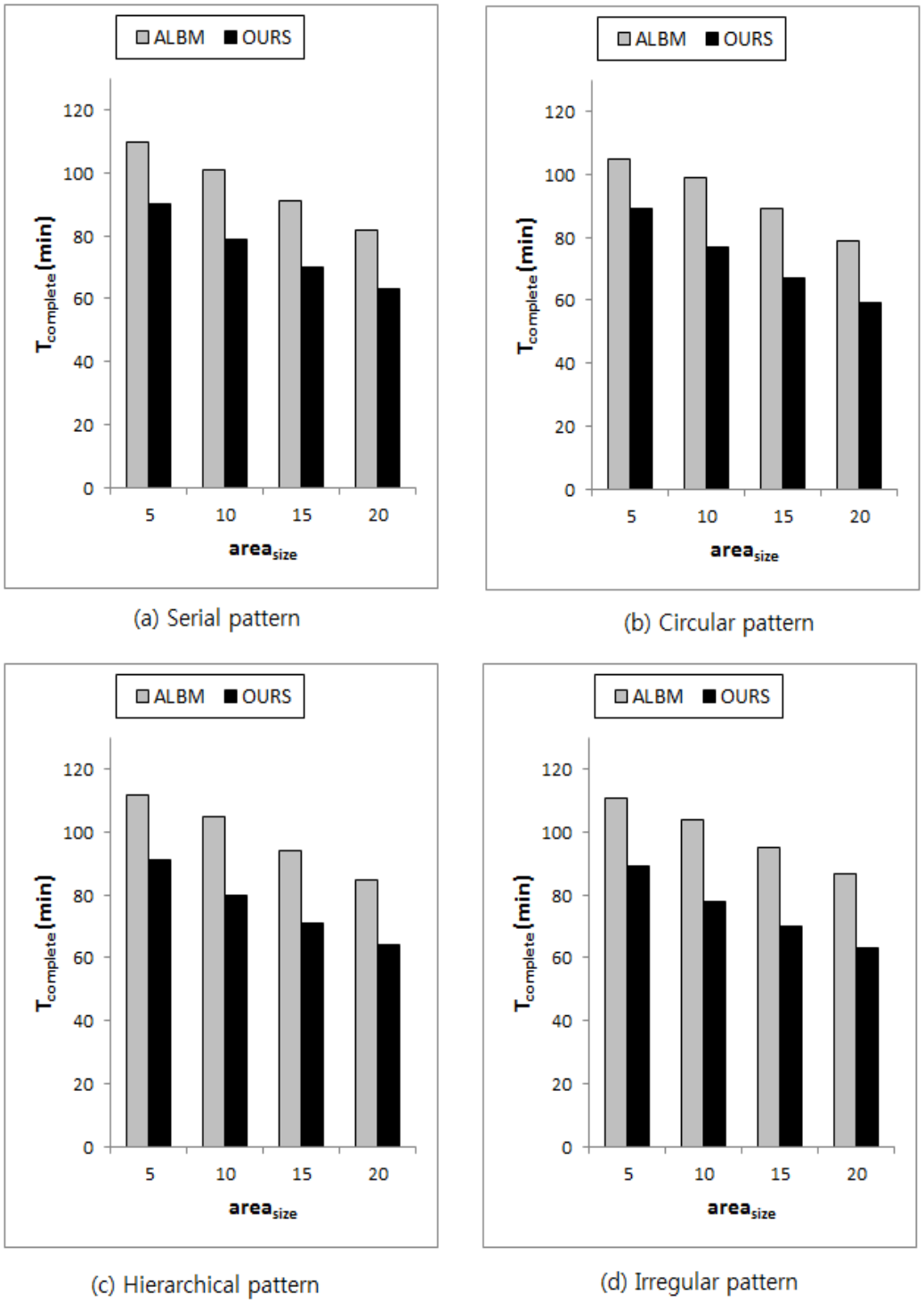


Figure 10. $T_{complete}$ for the Four Communication Patterns with Varying Values of $Area_{size}$

Acknowledgments

This work was supported by Kyonggi University Research Grant 2015(2015-035).

References

[1] G. R. Andrews, "Paradigms for process interaction in distributed programs", ACM Computing Surveys, vol. 23, no. 1, **(1991)**, pp. 49-90.
[2] J. Ahn, "Virtual Sender-based Message Logging for Large-scale Ubiquitous Sensor Network Systems", International Journal of Multimedia and Ubiquitous Engineering, vol. 9, no. 5, **(2014)**, pp. 37-48.
[3] R. Bagrodia, R. Meyer, M. Takai, Y. Chen, X. Zeng, J. Martin and H. Y. Song, "Parsec: A parallel simulation environment for complex systems," IEEE Computer, vol. 31, no. 10, **(1998)**, pp. 77-85.
[4] L. Bautista-Gomez, T. Ropars, N. Maruyama and S. Matsuoka, "Hierarchical clustering strategies for fault tolerance in large scale HPC systems", In Proc. of the IEEE International Conference on Cluster Computing, **(2012)**, pp. 355-363.
[5] W. Bland, A. Bouteiller, T. Herault, G. Bosilca and J. J. Dongarra, "Post-failure recovery of MPI communication capability: design and rationale", The International Journal of High Performance Computing Applications, vol. 27, no. 3, **(2013)**, pp. 244-254.
[6] A. Bouteiller, F. Cappello, T. Herault, G. Krawezik, P. Lemarinier and F. Magniette, "MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging", In Proc. of the International Conference on High Performance Networking and Computing, **(2003)**, pp. 1-17.
[7] D. Buntinasd, C. Coti, T. Herault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguez and F. Cappello, "Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI protocols", Future Generation Computer Systems, vol. 24, no. 1, **(2008)**, pp. 73-84.
[8] K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems", ACM Transactions on Computer Systems, vol. 3, no. 1, **(1985)**, pp. 63-75.
[9] F. Cappello, A. Guermouche and M. Snir, "On communication determinism in parallel HPC applications", In Proc. of the 19th International Conference on Computer Communications and Networks, **(2010)**, pp. 1-8.
[10] S. Di, L. Bautista-Gomez and F. Cappello, "Optimization of multi-level checkpoint model with uncertain execution scales", In Proc. of the International Conference for High Performance Computing, Networking, Storage, and Analysis, **(2014)**, pp. 907-918.
[11] T. T. Dinh, M. Lees, G. Theodoropoulos and R. Minson, "Large Scale Distributed Simulation of p2p Networks", In Proc. of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing, **(2008)**, pp. 499-507.
[12] E. Elnozahy, L. Alvisi, Y. Wang and D. Johnson, "A survey of rollback-recovery protocols in message-passing systems", ACM Computing Surveys, vol. 34, no. 3, **(2002)**, pp. 375-408.
[13] B. Gupta, R. Nikolaev and R. Chirra, "A recovery scheme for cluster federations using sender-based message logging", Journal of Computing and Information Technology, vol. 19, vo. 2, **(2011)**, pp. 127-139.
[14] P. Jaggi and A. Singh, "Log based recovery with low overhead for large mobile computing systems", Journal of Information Science and Engineering, vol. 29, no. 5, **(2013)**, pp. 969-984.
[15] D. Johnson and W. Zwaenpoel, "Sender-based message logging", In Proc. of the 7th International Symposium on Fault-Tolerant Computing, **(1987)**, pp. 14-19.
[16] L. Lamport, "Time, clocks, and the ordering of events in a distributed system", Communications of the ACM, vol. 21, no. 7, **(1978)**, pp. 558-565.
[17] 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, **(2009)**, pp. 124-133.
[18] H. F. Li, Z. Wei and D. Goswami, "Quasi-atomic recovery for distributed agents", Parallel Computing, vol. 32, no. 10, pp. 733-758, **(2009)**.
[19] Y. Luo and D. Manivannan, "HOPE: A Hybrid Optimistic checkpointing and selective Pessimistic mEssage logging protocol for large scale distributed systems", Future Generation Computer Systems, vol. 28, no. 8, **(2012)**, pp. 1217-1235.
[20] H. Meyer, D. Rexachs and E. Luque, "Hybrid Message Pessimistic Logging. Improving current pessimistic message logging protocols", Journal of Parallel and Distributed Computing, vol. 104, no. C, **(2017)**, pp. 206-222.

[21] M. Powell and D. Presotto, "Publishing: a reliable broadcast communication mechanism", In Proc. of the 9th International Symposium on Operating System Principles, **(1983)**, pp. 100-109.
[22] T. Ropars and C. Morin, "Active optimistic and distributed message logging for message-passing applications", Concurrency and Computation: Practice and Experience, vol. 23, no. 17, **(2011)**, pp. 2167-2178.
[23] R. D. Schlichting and F. B. Schneider, "Fail-stop processors: an approach to design-ing fault-tolerant distributed computing systems", ACM Transactions on Computer Systems, vol. 1, no. 3, **(1985)**, pp. 222-238.
[24] R. E. Strom and S. A. Yemeni, "Optimistic recovery in distributed systems", ACM Transactions on Computer Systems, vol. 3, no. 3, **(1985)**, pp. 204-226.
[25] G. Theodoropoulos, Y. Zhang, D. Chen, R. Minson, S. J. Turner, W. Cai, Y. Xie and B. Logan, "Large Scale Distributed Simulation on the Grid", In the 6th IEEE International Symposium on Cluster Computing and the Grid, vol. 2, **(2006)**, p. 63.
[26] J. Xu, R.B. Netzer and M. Mackey, "Sender-based message logging for reducing roll-back propagation", In Proc. of the 7th International Symposium on Parallel and Distributed Processing, **(1995)**, pp. 602-609.
[27] B. Yao, K. Ssu and W. Fuchs, "Message logging in mobile computing", In Proc. of the 29th International Symposium on Fault-Tolerant Computing, **(1999)**, pp. 14-19.

Authors

Jinho Ahn, he received his B.S., M.S. and Ph.D. degrees in Computer Science and En-gineering from Korea University, Republic of Korea, in 1997, 1999 and 2003, respectively. He has been a full professor in Department of Computer Science, Kyonggi University since 2003. 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.

