JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 33, 1575-1593 (2017)
DOI: 10.6688/JISE.2017.33.6.12

SBML Protocol for Conquering Simultaneous Failures with Group Dissemination Functionality*

JINHO AHN
Department of Computer Science
Kyonggi University
Suwon-si Gyeonggi-do, 16227 Korea
E-mail: jhahn@kgu.ac.kr

This paper presents a new sender based message logging (SBML) protocol to tolerate simultaneous failures by using the beneficial features of FIFO group communication links effectively. The protocol can lift the inherent weakness of the original SBML by replicating the log information of each message sent to a process group into the volatile storages of its members. Therefore, even if only one process in a group survives at a time, our protocol can progress the execution of the entire system without stopping and restarting it. Also, it needs no extra control message by piggybacking the additional information on the control message for logging every previous protocol essentially requires. The experimental results show our protocol can be a low cost solution for addressing the important drawback of the original SBML based on group communication without RSN replication functionality.

Keywords: distributed system, simultaneous failure, group communication, message logging, rollback recovery

1. INTRODUCTION

As the scale and the complexity of distributed and parallel applications rapidly grow in an unprecedented pace, the research on developing large-scale distributed computing system platforms is gaining a big attention in various application fields such as cloud computing, social networks, unmanned aerial systems, disaster recovery, smart power grid, and so on [16, 17, 23, 24]. These platforms should provide the users with an optimized and on-demand composition of services using highly dynamic, asynchronous, and geographically dispersed resources in a transparent way unlike the previous ones. A large-scale platform, that utilizes many low cost but powerful commodity computing devices, may have a higher chance of a failure as its components are more likely to crash than in small-scale platforms [4, 7, 10, 15, 18, 20]. Thus, fault-tolerance techniques should be employed to reduce the wasted execution time. For this purpose, one of two methods can be used, that is, 1) process replication, or 2) log-based rollback recovery. But the first method may degrade the scalability of the platform significantly due to its high overhead of synchronization among the replicated processes [11]. Thus, in this paper, we focus on the second method, log-based rollback recovery, due to its low overhead.

Among the log-based recovery techniques [1, 4, 5, 14, 19, 26], the sender-based message logging (SBML) with checkpointing [6, 12, 16, 25] is used commonly as a low-cost transparent rollback recovery technique in many fields such as mobile compu-

Received August 19, 2016; revised October 27, 2016; accepted January 9, 2017.
Communicated by Cho-Li Wang.
* This work was supported by Kyonggi University Research Grant 2014 (2014-011).

ting, cluster and grid computing, sensor network, and so on. This popularity is due to no need for specialized hardware and a considerably low overhead for synchronous logging on a stable storage by using volatile logging at sender's memory [6, 12, 16, 25]. However, every previous SBML protocol has the restriction that they can tolerate only a single failure at one time, called *sequential failures*. Thus, even if the protocol has been executed, simultaneous failures may cause the system inconsistency problem [1, 12].

As group-based computing has prevailed as a general model of current and future generation computing, multicast communication is becoming a mandatory building block for this computing. However, as all the existing SBML protocols have assumed reliable FIFO point-to-point communication links, they may not effectively employ log information of the same message to a process group its members received if they are applied into group communication link-based distributed system platforms. This paper presents a new sender-based message logging protocol to tolerate simultaneous failures by using the beneficial features of group communication links effectively. In order to address the critical weakness of the previous protocols, the proposed protocol makes every process know the respective receive sequence number of each same message to every other live group member by saving the number from the sender of the message onto its own volatile memory. This feature enables the protocol to survive a number of simultaneous failures except the whole system failure. Also, it needs no extra control message by piggybacking the additional information on the control message for logging every previous protocol essentially requires.

The remainder of the paper is structured as follows. In section 2, we describe the distributed system model assumed and, in section 3, the limitation of the previous SBML in detail. Section 4 introduces our SBML protocol and section 5 shows its correctness proof. In sections 6 and 7, we analyze and evaluate our protocol over the original one and, in section 8, conclude this paper.

2. SYSTEM MODEL

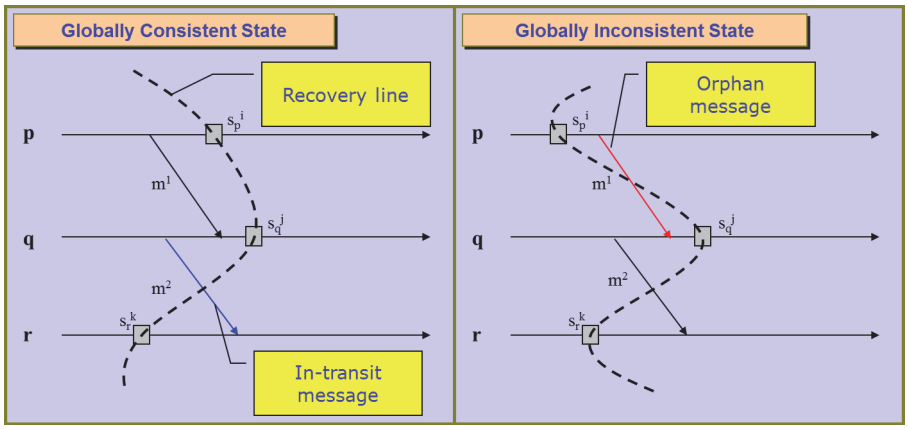
A distributed computation consists of a set P of $n(n > 0)$ sequential processes executed on sensor nodes in the system and there is a distributed stable storage that every process can always access that persists beyond processor failures, thereby supporting recovery from failure of an arbitrary number of processors [11]. 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 sensor 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 [21]. Events of processes occurring in a failure-free execution are ordered using Lamport's happened before relation [13], $\rightarrow^{hb}$, defined by the following three conditions. Let p, q and r be three processes $\in P$ and e^k_p be the kth event of $p(k > 0)$:

- If e_p^i and e_q^j occur, $p=q$ and $i<j$, then $e_p^i \rightarrow^{hb} e_q^j$.
- If e_p^i is the event that p sends a message to q, e_q^j is the event that q receives the message from p and then delivers it to the application, and $p \neq q$, then $e_p^i \rightarrow^{hb} e_q^j$.

• If $e_p^i \rightarrow^{hb} e_q^j$ and $e_q^j \rightarrow^{hb} e_r^k$, then $e_p^i \rightarrow^{hb} e_r^k$.

The main goal of log-based rollback recovery is to bring the system to a failure-free state when inconsistences occur due to failures. The execution of each process is piecewise deterministic [7, 22]: 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 kth state interval of process p, denoted by $si^k_p (k > 0)$, is started by the delivery event of the kth 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, consisting 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, Fig. 1 shows two examples of global states, which are shown by broken arrows. In Fig. 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 Fig. 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.

Fig. 1. Examples illustrating how to decide whether the state is globally consistent.

3. BACKGROUND

Before introducing our proposed SBML protocol to be capable of overcoming the limitations of the previous SBML, let us identify the exact reasons why the latter has its incapability against tolerating simultaneous failures based on both unicast and group communication links using some examples. Originally, sender-based message logging [12] is designed to have the positive feature of receiver-based pessimistic message logging [19, 26], *no roll-back property*, in case of sequential failures. Also, it may significantly reduce the 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 feature mentioned above using Fig. 2. In the figure, three processes, $p0$, $p1$, and $p2$, execute their respective computation together by exchanging messages with each other. Processes $p0$ and $p2$ send messages m^1 and m^2 to process $p1$ respectively. In this operation, each sender 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 denoted by $SLog(m)$, which 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 [1, 12, 22, 25]. Here, *partially logged* means the *RSN* of the message has not been recorded on $SLog(m)$ yet. Then, $p1$ first receives message m^1 from $p0$, increments its *RSN*, RSN_1, by one, and assigns the value of RSN_1 ($=\alpha$) to the message. Next, $p1$ sends sender $p0$ an acknowledgment message including m^1's *RSN*. Similarly, when $p1$ receives m^2 from $p2$, it performs the same procedure, where the assigned value of m^2's *RSN* is (α+1) in this example. When each sender, *e.g.*, $p0$ or $p2$, obtains an acknowledgment message for its sent message m, *e.g.*, m^1 or m^2, from the corresponding receiver, it saves m's *RSN* attached to the acknowledgment into m's log information, $SLog(m)$. At this time, m is called *fully logged*, meaning every element in $SLog(m)$ is filled with its actual value for m's recovery. Then, the sender notifies $p1$ of the fact that it safely holds the full log information on its own volatile memory. So, even if $p1$ fails afterwards, it can get the full log information for m^1 and m^2 from the two senders respectively and replay them in the same order like in the pre-failure state. Therefore, even though there have been any messages sent from $p1$ after m^1 and m^2 in this case, they would not become orphan messages.

Let us consider what happens when several processes fail at the same time using Fig. 3. This figure shows an execution similar to the one in Fig. 2 except that $p3$ joins the execution and sends the third message m^3 to $p1$. After all the logging procedures for the three messages have been completed, $p1$ transmits message m^4 to $p2$. Then, suppose the three processes $p0$, $p1$ and $p3$ crash simultaneously. In this case, as $p0$ and $p3$ lost the values of the *RSN*s of messages m^1 and m^3 on their respective volatile memories due to the failures, they cannot provide the *RSN* values for $p1$ during recovery.

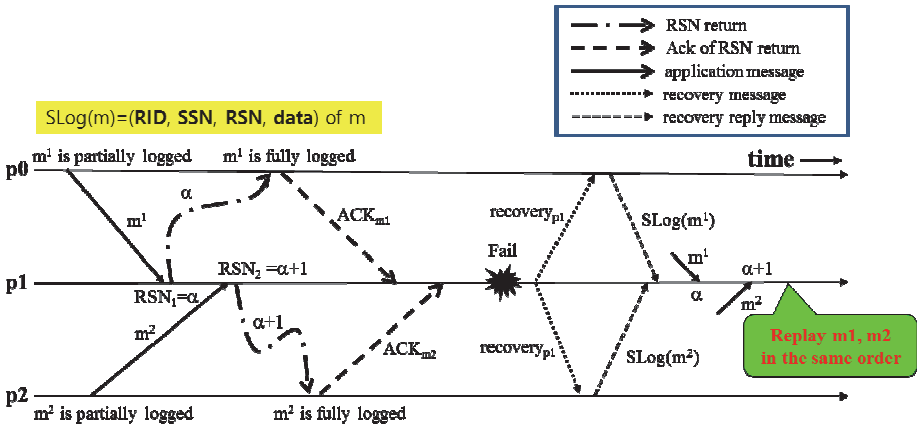


Fig. 2. An illustration showing how the original SBML to address sequential failures.

Also, $p2$ has only the value of m^2's RSN. This log information deficiency causes the following ambiguous situation: $p1$ cannot determine in which order m^1, m^2 and m^3 should be replayed. Thus, $p1$ may not reconstruct message m^4 during recovery, which makes $p2$ orphan process. Due to this incapability, every previous sender-based message logging may not make sure the entire system consistency on simultaneous failure occurrences. However, if unicast-only communication links are assumed in the system model, this technique would be unavoidably destined to this limitation without the help of any other compensating methods [1, 12].

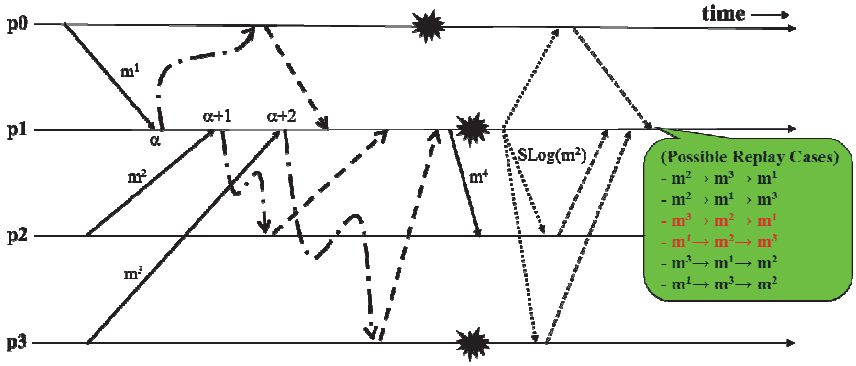


Fig. 3. An illustration showing why the original SBML cannot address simultaneous failures.

Suppose group dissemination functionality is the basic communication feature of the system. With this assumption, we could naively anticipate $p1$ can get m^1's *RSN* from $p2$ even if both $p0$ and $p1$ crash simultaneously in Fig. 2 because $p2$ also received m^1 sent to the group consisting of $p0$, $p1$ and $p2$. Let us identify that the original SBML can satisfy this expectation using Fig. 4. In reality, actual process group size may vary depending on application types, but for the sake of simplicity of explanation, we assume a very small group in the examples used later. In the figure, there are three processes, p_g^0, p_g^1 and p_g^2, composing a process group g, and there are three messages, m_g^1, m_g^2 and m_g^3,

sent to group g from senders S_1, S_2 and S_3 respectively. Here, assume each sender may also be a member of group g. In this example, suppose the RSNs of p_g^0, p_g^1 and p_g^2 be (α–1), (β–1), (γ–1) ($\alpha \neq \beta \neq \gamma$) in order before sending out the three messages. Here, all three messages will eventually be delivered to every process member through reliable FIFO group communication links, but the RSNs of each message assigned by all three processes may all be different due to delivery order asynchrony of messages sent to a group from different senders. In this example, p_g^0, p_g^1 and p_g^2 receive m^1, m^2 and m^3, m^2, m^3 and m^1, and m^3, m^1 and m^2 in order, respectively. In this case, the *RSN*s they have assigned to m^1 could be α, β+2 and γ+1 like in this figure. Thus, if the sender of m^1, S_1, and p_g^0 crash at the same time, m^1 may not be replayed with its pre-failure *RSN* value, α, at p_g^0 during recovery.

For example, Fig. 5 shows the case that the senders of the three messages are p_g^0, p_g^1 and p_g^2, respectively. Here, p_g^0 receives m^1, m^2 and then m^3 whose *RSN*s become each

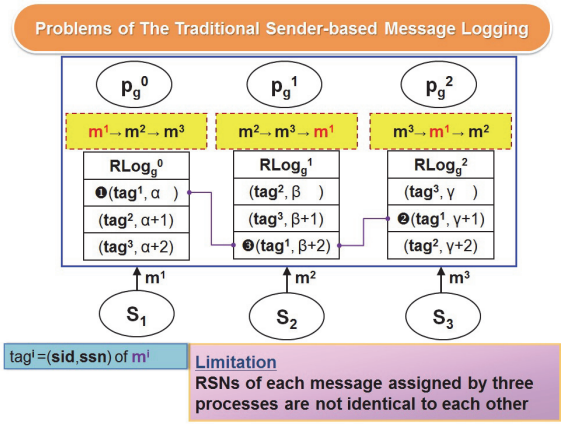


Fig. 4. Limitation of the original SBML based on reliable FIFO group communication links.

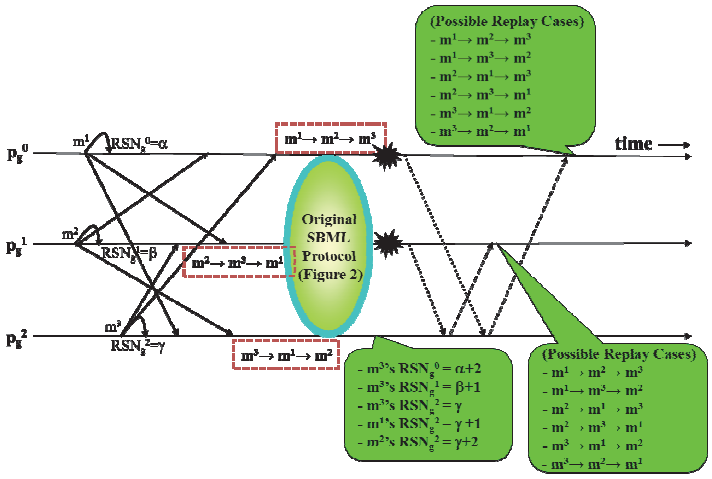


Fig. 5. An execution case showing the inconsistency problem the original SBML based on group dissemination links may incur in case of simultaneous failures.

α, (α+1) and (α+2). Similarly, p_g^1 assigns β, (β+1) and (β+2) as *RSN* to m^2, m^3 and m^1 and p_g^2, γ, (γ+1) and (γ+2), m^3, m^1 and m^2. Afterwards, every message receipt event on each process in this figure triggers the semi-synchronous logging procedure for the corresponding message like in Fig. 2. Hereafter, suppose p_g^0 and p_g^1 crash at the same time. During recovery, they can get the values of m^3's *RSN*s from p_g^2, (α+2), (β+1) and γ, assigned by each process, but no information about the values of m^1's and m^2's *RSN*s they allocated in the pre-failure state. Thus, neither p_g^0 nor p_g^1 can decide in which order the three messages should be replayed. If there were any messages either p_g^0 or p_g^1, or both sent to p_g^2 after the completion of logging m^1, m^2 and m^3 before their failures, p_g^2 might be an orphan process after performing the recovery procedure of the original SBML.

4. NOVEL SBML PROTOCOL

From the observation, it is found out that as the original sender-based message logging has been developed assuming reliable FIFO unicast communication functionality, it could not utilize the advantageous features of the group communication functionality, which may potentially make a breakthrough to overcome its annoying constraint. Thus, our SBML protocol is designed to have the following beneficial features to address it;

- Replicate the *RSN* information of a message sent to a group, separately assigned by each member, into volatile storages of other group members.
- Force the state made after each message multicast to a group to be unable to be visible to any other process until the replication has been completed.

With these features, the protocol can tolerate simultaneous failures while it attempts to minimize additional inter-process communication cost required for no rollback of live processes by piggybacking the additional information on the control message for logging every previous protocol essentially needs. For this purpose, the data structures each group member should maintain in our protocol are following;

- SSN_g^i: the send sequence number of the latest message sent by p_g^i.
- RSN_g^i: the receive sequence number of the latest message delivered to p_g^i.
- $SSNVt_g^i$: a vector where $SSNVt_g^i[q]$ is the *SSN* of the last message that was delivered to p_g^i from a process p_g^q.
- $XSLog_g^i$: a set saving *e*(*gid*, *ssn*, *rsnlist*, *data*) of each message sent by p_g^i. Here, *e* is the log information of a message and the first two fields and the last field are the identifier of the process group, the send sequence number and data of the message respectively. The third field is the list of receive sequence numbers of the message which all receivers of the message assigned to it. Its element consists of a pair of (*pid*, *rsn*), where *pid* is one of the receivers and *rsn* is the *RSN* of the message *pid* assigns to it.
- $XRLog_g^i$: a set which maintains *e*(*sid*, *ssn*, *rsnlist*) of each message received by p_g^i. Here, *e* is the log information of the message and the first two fields are the sender's *id* and the send sequence number of the message respectively. The last field is a set of elements whose form is (*pid*, *rsn*) where *pid* is one of group members including p_g^i that has assigned *rsn* as *RSN* to the message on its receipt. This set can help other crashed

group members perform replaying the message in their pre-failure orders with its corresponding *RSN*.

- $stableRSN_g^i$: the receive sequence number of the latest message which has been delivered to p_g^i and replicated on the volatile storage of every member or checkpointed on the stable storage. It is used for indicating until which messages p_g^i can send to other processes.

In order to satisfy the requirements mentioned above, our protocol is performed as follows. For this purpose, let us closely look at an example showing the detail of execution of the protocol. This example is the same as that of Fig. 5. In Fig. 6, processes p_g^0, p_g^1 and p_g^2, multicast three messages, m^1, m^2 and m^3, to every member of group g including itself respectively. Here, p_g^0 receives m^1, m^2 and then m^3 whose *RSN* values become α, (α+1) and (α+2) in order. When p_g^1 receives m^2, m^3 and m^1 in order, it assigns β, (β+1) and (β+2) as *RSN* to them respectively. Similarly, after p_g^2 has received m^3, m^1 and m^2 in order, their *RSN*s become γ, (γ+1) and (γ+2) respectively. All the three processes execute the proposed protocol like in Fig. 7. For simplicity, this figure only shows the case p_g^0 disseminates m^1 to every group member including itself. First, after sending m^1, p_g^0 saves the partial log element (tag_g^1, $\varnothing$, $data_{m1}$) into $XSLog_g^0$ in procedure **Module G-SEND(*data*$_{m1}$, *g*)** in Fig. 8. When having received m^1, p_g^0, p_g^1 and p_g^2 add their respective receiver log elements, (tag_g^1, {(p_g^0, α)}), (tag_g^1, {(p_g^1, β+2)}) and (tag_g^1, {(p_g^2, γ+1)}), to $XRLog_g^0$, $XRLog_g^1$ and $XRLog_g^2$ in **Module G-RECV(*tag*$_g^1$, *data*$_{m1}$)**. Then, they send each an acknowledgment, *rsn-return*(tag_g^1, α), *rsn-return*(tag_g^1, β+2) and *rsn-return*(tag_g^1, γ+1), to m^1's sender p_g^0. When p_g^0 receives the three return messages in a particular order, it inserts their *RSN* elements, (p_g^0, α), (p_g^1, β+2) and (p_g^2, γ+1), into $XSLog_g^0$ in **Module RCV-RSN(*tag*$_g^1$, *RSN*$_{m1}$)**. When p_g^0 has collected each an acknowledgment from every live group member, it multicasts a control message including the value of the *RSN* of m^1 assigned by the member, *e.g.*, {(p_g^0, α), (p_g^1, β+1), (p_g^2, γ)}, to the group. When the control message has arrived, p_g^0, p_g^1 and p_g^2 update their receiver logs, $XRLog_g^0$, $XRLog_g^1$ and $XRLog_g^2$, with the list of *RSN*s piggybacked on the message in **Module RCV-ACK(*tag*$_g^1$, *RSNS*$_{m1}$)**. In Fig. 6, m^2 and m^3 also experience the same

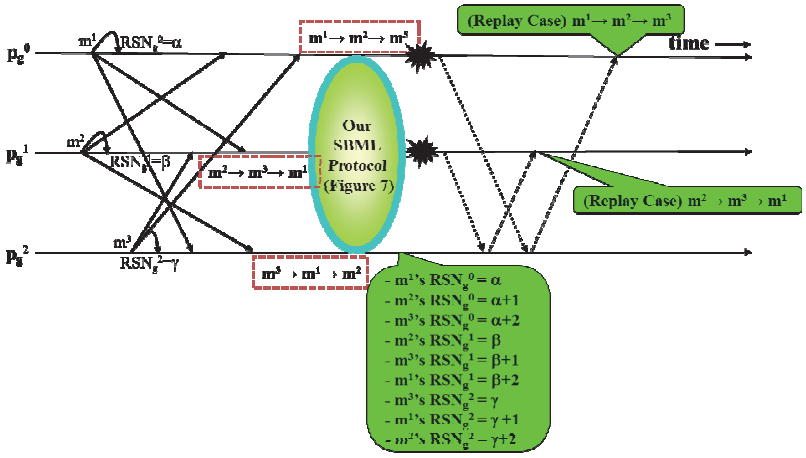


Fig. 6. Illustration showing how our SBML protocol can effectively address simultaneous failures.

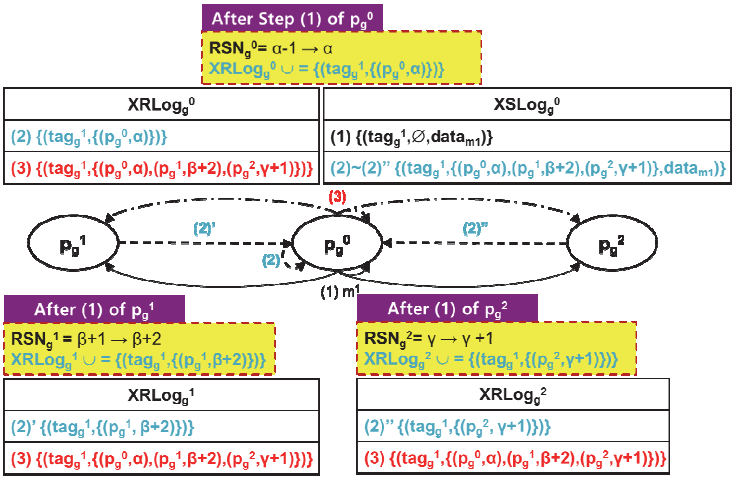


Fig. 7. An illustration showing the detail of our SBML protocol based on group dissemination links with message m^1 multicast to group g from process p_g^0.

logging procedures as those of Fig. 7. In this way, after our protocol has been completed with the three messages, every group member can maintain the values of all the RSNs assigned to each message by the member, *e.g.*, (tag_g^1, {(p_g^0, α), (p_g^1, β+2), (p_g^2, γ+1)}), (tag_g^2, {(p_g^0, α+1), (p_g^1, β), (p_g^2, γ+2)}) and (tag_g^3, {(p_g^0, α+2), (p_g^1, β+1), (p_g^2, γ)}), on its volatile memory.

Thanks to this log information replication, even though p_g^0 and p_g^1 fail at the same time in Fig. 6, they trigger **Module RECOVERY()** in Fig. 8 to be able to obtain their own RSN values for every received message from **Module RCV-RQTRCVY(p_g^i)** of p_g^2 during recovery. Then, p_g^0 can deterministically replay m^1, m^2 and m^3 in order, and p_g^1, m^2, m^3 and m^1 like before failure. Therefore, even if p_g^0 or p_g^1 transmitted any message to p_g^2 after m^1, m^2 and m^3 in its pre-failure state, p_g^2 would not be orphan process after recovery.

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

5. CORRECTNESS PROOF

This section proves our proposed SBML protocol ensures system consistency in case of concurrent failures using one lemma and one theorem.

Lemma 1: After our protocol has been completed, no crashed process P_g^i in group g makes the state of any other surviving one in group g inconsistent.

***Proof*:** Suppose the sequence of all the messages fully logged whose receiver was P_g^i in its pre-failure state, denoted by $SEQ\text{-}FLMSGs_g^i$. The proof proceeds by induction on the number of all the messages in $SEQ\text{-}FLMSGs_g^i$, denoted by $NUMOF(SEQ\text{-}FLMSGs_g^i)$.

[Base case] In this case, P_g^i received only one fully logged message m to group g in its pre-failure state and two cases should be considered.

Case 1: m's sender P_g^s is alive.

In this case, P_g^i can trivially obtain all *RSN*s of m from $XSLog_g^s$ of P_g^s all the other processes in group g including P_g^i have assigned to m before. Thus, P_g^i can record them into $XRLog_g^i$ and then, replay m in its original order like before failure. Therefore, no live process having received any message whose sender is P_g^i after m's receipt becomes orphan.

Case 2: m's sender P_g^s is a crashed process.

In this case, two sub-cases should be considered.

Case 2.1: No live process in group g exists

In this case, P_g^i replaying m in an order different from before failure doesn't make the global system state inconsistent because there is no orphan state on which m's receive event occurrence before its failure has any impact.

Case 2.2: At least one live process P_g^j in group g exists.

In this case, as message m has been fully logged, all live processes including P_g^j already obtained all *RSN*s of m from $XSLog_g^s$ of P_g^s all the other processes in group g including P_g^i had assigned to m before failure. Thus, P_g^i can get the *RSN*s from $XRLog_g^j$ of P_g^j and record them into $XRLog_g^i$. Then, recovered P_g^s provides m's data for P_g^i, which can replay m in its original order like before failure. Therefore, no live process having received any message whose sender is P_g^i after m's receipt becomes orphan.

//When process p_{gid}^i attempts to multicast a message *m* including data to every member of group *gid*.//

Module G-SEND(*data*, *gid*) AT p_{gid}^i

$SSN_{gid}^i \leftarrow SSN_{gid}^i + 1$; **multicast** $m(p_{gid}^i, SSN_{gid}^i, data)$ **to** group *gid*;

$XSLog_{gid}^i \leftarrow XSLog_{gid}^i \cup \{(gid, SSN_{gid}^i, \varnothing, data)\}$;

//When process p_{gid}^i receives a message *m* from message sender *m.sndr*.//

Module G-RECV($m(sndr, ssn, data)$) AT p_{gid}^i

if($SsnVt_{gid}^i[m.sndr] < m.ssn$) **then**

$RSN_{gid}^i \leftarrow RSN_{gid}^i + 1$;

$SsnVt_{gid}^i[m.sndr] \leftarrow m.ssn$;

send *rsn-return*(p_{gid}^i, *m.ssn*, RSN_{gid}^i) **to** *m.sndr*;

$XRLog_{gid}^i \leftarrow XRLog_{gid}^i \cup \{(m.sndr, m.ssn, \{(p_{gid}^i, RSN_{gid}^i)\})\}$;

delay all the send message operations generated after having received *m*;

deliver *m.data* **to** its corresponding application;

else

find $\exists e \in XRLog_{gid}^i$ **st** $((e.sndr = m.sndr) \wedge (e.ssn = m.ssn))$;

find $\exists o \in e.rsnlist$ **st** $(o.rcvr = p_{gid}^i)$;

send *rsn-return*(p_{gid}^i, *m.ssn*, *o.rsn*) **to** *m.sndr*;

//When message sender p_{gid}^i receives a message *rsn-return* from the receiver *rsn-return.rcvr* of a message whose *RSN* value is *rsn-return.rsn*.//

Module RCV-RSN(*rsn-return*(*rvcr*, *ssn*, *rsn*)) AT p_{gid}^{i}
  find $\exists e \in XSLog_{gid}^{i}$ **st** ((*rsn-return.rcvr* $\in$ *e.gid*) ^ (*e.ssn* = *rsn-return.ssn*)) ;
  if($\neg$($\exists o \in$ *e.rsnlist* **st** (*rsn-return.rcvr* = *o.rcvr*))) **then**
    e.rsnlist $\leftarrow$ *e.rsnlist* $\cup$ {(*rsn-return.rcvr*, *rsn-return.rsn*)};
    if(p_{gid}^{i} has received each a *RSN* from every other live member $p_{gid}^{k} \in$ *e.gid*) **then**
      multicast *ack*(p_{gid}^{i}, *ssn*, *e.rsnlist*) **to** *e.gid*;
  else if(p_{gid}^{i} has received each a *RSN* from every other live member $p_{gid}^{k} \in$ *e.gid*) **then**
      send *ack*(p_{gid}^{i}, *ssn*, *e.rsnlist*) **to** *rsn-return.rcvr*;

//When message receiver p_{gid}^{i} receives an acknowledgement *ack* from message sender *ack.sndr* indicating the latter has fully logged the corresponding message in its volatile storage and collected the list of *RSN*s all other live group members have each assigned for the message.//
Module RCV-ACK(*ack*(*sndr*, *ssn*, *rsnlist*)) AT p_{gid}^{i}
  find $\exists e \in$ *ack.rsnlist* **st** (*e.rcvr* = p_{gid}^{i});
  if($stableRSN_{gid}^{i} <$ *e.rsn*) **then**
    find $\exists o \in XRLog_{gid}^{i}$ **st** ((*o.sndr* = *ack.sndr*) ^ (*o.ssn* = *ack.ssn*));
    o.rsnlist $\leftarrow$ *ack.rsnlist*;
    allow all the send message operations delayed before receiving the message whose *RSN* value is (*e.rsn* +1) **to** begin executing;
    $stableRSN_{gid}^{i} \leftarrow$ *e.rsn*;

//When process p_{gid}^{i} takes its local checkpoint on the stable storage.//
Module TAKE-CHECKPOINT() AT p_{gid}^{i}
  take its local checkpoint with (RSN_{gid}^{i}, SSN_{gid}^{i}, $SsnVt_{gid}^{i}$, $XSLog_{gid}^{i}$) **on** the stable storage;
  allow all the send message operations delayed before this checkpoint **to** begin executing;
  $stableRSN_{gid}^{i} \leftarrow RSN_{gid}^{i}$;

Fig. 8. Our group-based SBML procedures during failure-free operation.

//When process p_{gid}^{i} attempts to recover after failure. //
Module RECOVERY() AT Recovering Process p_{gid}^{i}
  restore a latest checkpointed state with (RSN_{gid}^{i}, SSN_{gid}^{i}, $SsnVt_{gid}^{i}$, $XSLog_{gid}^{i}$) **from** stable storage;
  multicast each a recovery request *rqt-rcvy*(p_{gid}^{i}) **to** group *gid*;

  //p_{gid}^{i} collects all recovery information of fully or partially logged messages from the other live processes.//
  while recovery replies aren't received from all other group members **do**
    put fully logged messages for p_{gid}^{i} piggybacked on each reply *rpy-rcvy* **into** $flog_{gid}^{i}$ in *RSN* order;
    put partially logged messages for p_{gid}^{i} piggybacked on each reply *rpy-rcvy* **into** $plog_{gid}^{i}$ in FIFO order;

```
//p_gid^i replays every fully logged message in flog_gid^i in its RSN order like in p_gid^i's
pre-failure state.//
for all e ∈ flog_gid^i st (∃o ∈ e.rsnlist: (o.rcvr = p_gid^i) ^ (o.rsn = RSN_gid^i+1)) do
   RSN_gid^i ← RSN_gid^i +1;
   SsnVt_gid^i[e.sndr] ← e.ssn;
   XRLog_gid^i ← XRLog_gid^i ∪ {(e.sndr, e.ssn, e.rsnlist);
   deliver e.data to its corresponding application;
   flog_gid^i ← flog_gid^i −{e};
stableRSN_gid^i ← RSN_gid^i;
//p_gid^i replays every partially logged message in plog_gid^i in FIFO order like in
p_gid^i's pre-failure state.//
while plog_gid^i is a non-empty set do
   randomly select ∃e in plog_gid^i st (e.ssn = SsnVt_gid^i[e.sndr]+1);
   call Module G-RECV(e.sndr, e.ssn, e.data) at p_gid^i;
   plog_gid^i ← plog_gid^i −{e};

//When a surviving process p_gid^i receives a recovery message rqt-rcvy from anoth-
er process rqt-rcvy.rcvr requesting the log information for recovery of every pro-
cess including the latter from p_gid^i's volatile storage.//
Module RCV-RQTRCVY(rqt-rcvy(rcvr)) AT Live Process p_gid^i
   put fully and partially logged messages for rqt-rcvy.rcvr in XSLog_gid^i and XRLog_gid^i
      into a reply rpy-rcvy;
   send rpy-rcvy to rqt-rcvy.rcvr;
```

Fig. 9. Recovery and its assisting procedures.

[Induction hypothesis] We assume that the theorem is true for P_g^i in case that *NUMOF* ($SEQ–FLMSGs_g^i$)=k.

[Induction step] By induction hypothesis, P_g^i can obtain all the log information of k fully logged messages it has received before its failure. Therefore, if P_g^i may get the full log information of the (k+1)th message during recovery, the theorem is true for P_g^i in case *NUMOF* ($SEQ–FLMSGs_g^i$)=k+1. The following case is similar to the base case stated above.

By the induction, even if there are one or more crashed processes in group g, the proposed protocol can always keep the state of any other surviving one in its group consistent.

Theorem 1: The proposed protocol enables the global system state to be kept consistent despite simultaneous failures.

***Proof*:** We prove this theorem by contradiction. Assume that the protocol may not ensure to successfully complete tolerating k simultaneous failures. Suppose the set of all crashed processes in group g is denoted by $CRASHED_g$ and the set of all live processes in group g, $LIVE_g$. Two cases should be considered as follows:

Case 1: every process in group g crashes.
In this case, even if the process replays each received message in any particular order different from before failure, the global system state is always consistent.

Case 2: At least one process in group g is alive.
In this case, there may be one or more orphan state intervals having direct or indirect impact on their current states. Two sorts of orphan state intervals are following: direct state intervals starting with each message any crashed process has transmitted, $DIRECT\text{-}INTERVALs_g$, and indirect state intervals created by each message sent by any other live one, $INDIRECT\text{-}INTERVALs_g$.
In this case, two sub-cases should be considered:

Case 2.1: Any state interval $si \in DIRECT\text{-}INTERVALs_g$ starts with $receive_g(m)$.
In this case, suppose si is created by $receive_g^i(m)$ at $P_g^i \in LIVE_g$ and depends on the receive events of all the messages P_g^i has received until generating si including $receive_g^i(m)$. Even though all the senders of the received messages, denoted by $DSENDPROCs_g(si)$, would be a subset of $CRASHED_g$, by Lemma 1, si never becomes an orphan state because the proposed protocol forces no crashed process $\in DSENDPROCs_g(si)$ to make P_g^i be an orphan process.

Case 2.2: Any state interval $si \in INDIRECT\text{-}INTERVALs_g$ starts with $receive_g(m)$.
In this case, if si depends transitively on any state interval $si' \in DIRECT\text{-}INTERVALs_g$, it may be an orphan state if si' could not be restored even after having completed the recovery procedure. However, this situation cannot occur according to case 2.1.

Therefore, the globally consistent system state can always be kept consistent in case of simultaneous failures. This contradicts the hypothesis.

6. ANALYSIS

This section presents some numerical analysis results to compare our proposed SBML protocol (*OURS*) to the original one with *RSN* replication functionality for ensuring no rollback of surviving processes (*ORIGIN-REP*) [6, 12, 16, 25] regarding control message exchange overhead during failure-free operation. Here, as *ORIGIN-REP* is designed based on unicast communication links for enabling *RSN* replication, if a group size is k, it requires that k individual control messages from each message sender should be sent to all group members.

For this purpose, several parameters used are defined as follows:

- N_{proc}: the total number of processes in a group.
- N_{appmsg}: the total number of application messages generated in a group.
- C_{multi}: the cost of sending a multicast message to every member in a group.
- C_{uni}: the cost of sending a unicast message to an individual process.

In this evaluation, we assume a computer cluster of N_{proc} processes or nodes with two different one-way message costs, C_{uni} and C_{multi}. With this assumed model, the total control message costs of *ORIGIN-REP* and *OURS* occurring during failure-free opera-

tions, denoted by $ORIGIN\text{-}REP_{msg\text{-}cost}$ and $OURS_{msg\text{-}cost}$, can be expressed as Eqs. (1) and (2) respectively.

$$ORIGIN\text{-}REP_{msg\text{-}cost} = 2 * C_{uni} * (N_{proc} - 1) * N_{appmsg} \quad (1)$$

$$OURS_{msg\text{-}cost} = (C_{uni} * (N_{proc} - 1) + C_{mult}) * N_{appmsg} \quad (2)$$

The difference of control message costs of *ORIGIN-REP* and *OURS*, $\Delta DIFF_{msg\text{-}cost}$(= Eq. (1) − Eq. (2), is Eq. (3).

$$\Delta DIFF_{msg\text{-}cost} = (C_{uni} * (N_{proc} - 1) * N_{appmsg} \quad (3)$$

From Eq. (3), we can see that $\Delta DIFF_{msg\text{-}cost}$ may linearly become larger as the number of messages generated increases and the difference between C_{mult} and C_{uni} decreases. In general, a multicast sending(C_{mult}) is highly more efficient than achieving an equivalent job using unicast only send primitive(C_{uni}*(N_{proc}−1)) in most LAN or WAN-based multicast protocols developed in network or application layers. Thus, as N_{proc} increases, $\Delta DIFF_{msg\text{-}cost}$ may also be higher.

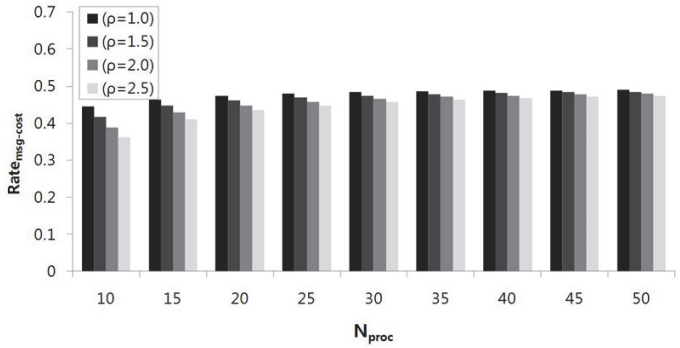


Fig. 10. $Rate_{msg\text{-}cost}$ with varying values of N_{proc} and ρ.

Let us clarify how much *OURS* may reduce the control message overhead of the entire system during failure-free operation compared with *ORIGIN-REP* using Figs. 10 and 11. The two figures show the variation of reduction rate, $Rate_{msg\text{-}cost}$, of $OURS_{msg\text{-}cost}$, against $ORIGIN\text{-}REP_{msg\text{-}cost}$ with varying N_{proc} ranging from 10 to 50 in case ρ(=C_{mult} / C_{uni}) is 1.0 through 4.5 at 0.5 intervals. The reduction rate is expressed as $\Delta DIFF_{msg\text{-}cost}$ / $ORIGIN\text{-}REP_{msg\text{-}cost}$. In these figures, as N_{proc} becomes bigger, their $Rate_{msg\text{-}cost}$s appear to be converging very close to 0.5. Especially, as ρ becomes close to 1, *i.e.*, the difference between two one-way communication costs, C_{mult} and C_{uni}, is smaller, the $Rate_{msg\text{-}cost}$ begins from much higher value. This outcome arises from the reason that the increase of the number of processes in a group and the decrease of ρ allows *OURS* to enormously lower the number of control messages between processes in a group during failure-free operation compared with *ORIGIN-REP*. In particular, if most of the physical network types used in a cluster are broadcast, *OURS* can reduce very close to 50% of the total control message cost of *ORIGIN-REP*.

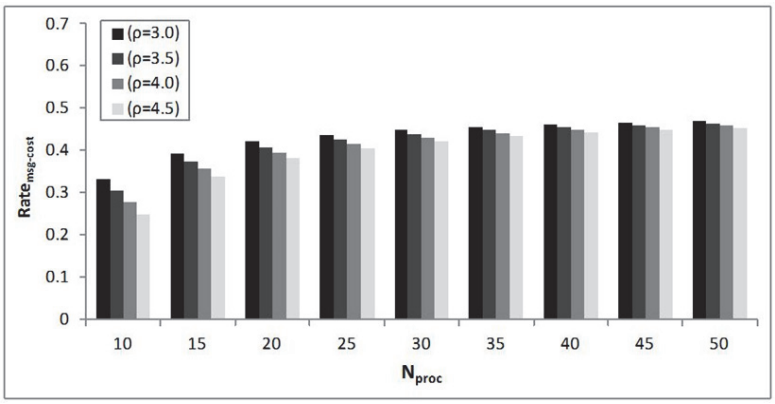


Fig. 11. $Rate_{msg\text{-}cost}$ with varying values of N_{proc} and ρ (continued).

In conclusion, these results show that our SBML protocol using the beneficial features of FIFO broadcast links remarkably lowers the cost of additional inter-process communication of the original SBML required for ensuring no rollback of surviving processes while being capable of tolerating concurrent failures without any inconsistency problem.

7. EVALUATION

In this section, we perform extensive simulations to measure the *RSN* replication overhead of our proposed SBML protocol (*OURS*) against the original one based on group communication without *RSN* replication functionality (*ORIGIN-GC-NOREP*) using a discrete-event simulation language [3]. Two performance indicators are used for comparison; the elapsed time until the same distributed execution has been completed ($T_{complete}$), and the increasing rate of the performance overhead of the two protocols for simultaneous failures against the original SBML, *ORIGIN*, for only tolerating a single failure at a time ($Increase_{Overhead}$). A system with N nodes connected through a general network is simulated. Each node has one process executing on it and, for simplicity, the processes are assumed to be initiated and completed together. Each process group consists of three processes and the number of process groups (*NOG*) is 10, 15, 20, 25, 30 and 35 (N=30, 45, 60, 75, 90, 105). Here, the degree of *RSN* redundancy of *OURS*, k, is configured to 3. The target of each message sent from a process is always a process group. Thus, IP multicast is used for multicasting a message to a group of processes. The message transmission capacity of a link in the network is 100Mbps and its propagation delay is 1ms. Every process has a 128MB buffer space for storing its message log. The message size ranges from 1KB to 1MB. Normal checkpointing is initiated at each process with an interval following an exponential distribution with a mean T_{nc}=300 seconds. In addition, a message to a process group is sent from a randomly chosen process with an interval following an exponential distribution with a mean of T_{ms}=3 seconds. 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 [2].

- 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.

In Fig. 12, $T_{complete}$ is indicated for the two protocols, *OURS* and *ORIGIN-GC-NOREP*, for each application interaction pattern when the number of process group

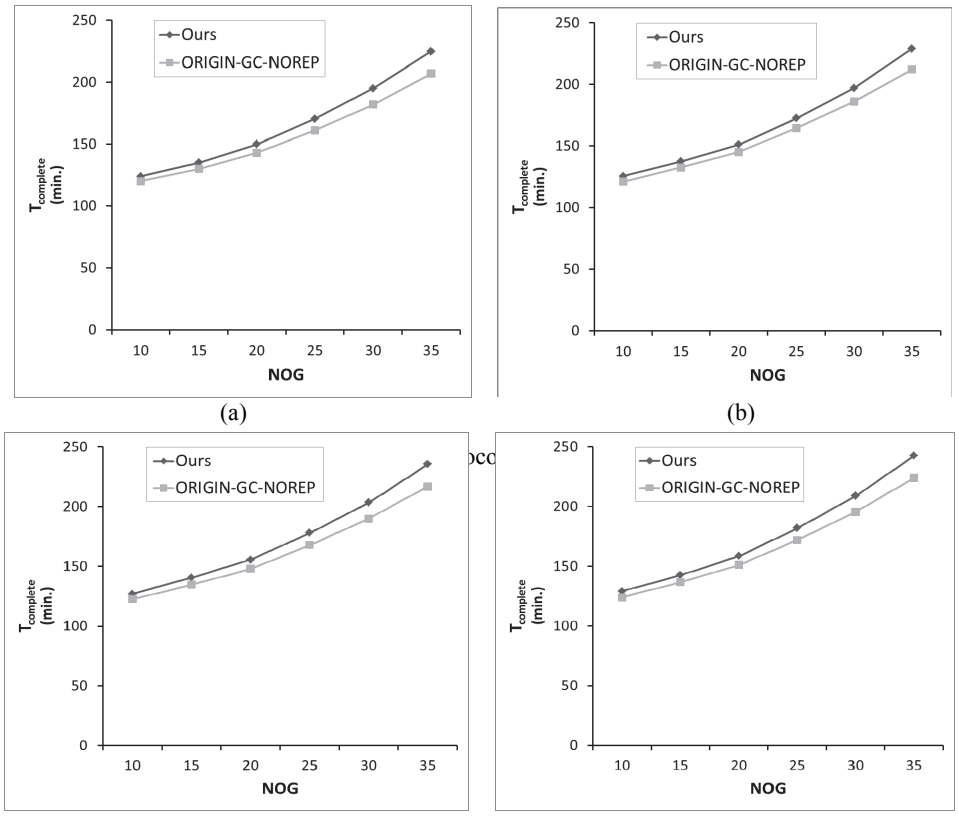


Fig. 12. Comparisons of $T_{complete}$ of the two protocols with varying values of *NOG*: (a) Serial pattern; (b) Circular pattern; (c) Hierarchical pattern; (d) Irregular pattern.

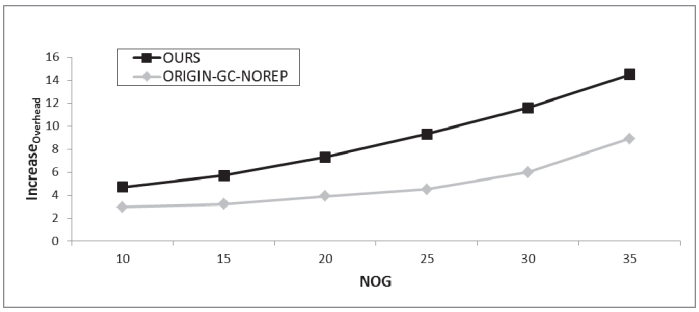


Fig. 13. Comparisons of $Increase_{Overhead}$ of the two protocols with varying values of *NOG*.

(*NOG*) changes from 10 to 35 whose size is 3. As *NOG* becomes bigger, $T_{complete}$ of the two protocols rise up accordingly because their inter-group communication costs are higher. Also, $T_{complete}$ of *OURS* is larger than $T_{complete}$ of *ORIGIN-GC-NOREP* because the *RSN* delivery time of the first may be longer during its *RSN* replication procedure compared with the latter. However, this simulation results indicate the difference between $T_{complete}$s of the two protocols ranges only from 3.2% through 7.9% of the latter irrespective of application interaction patterns.

8. CONCLUSIONS

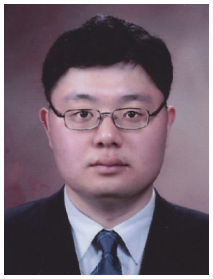
In group communication link-based distributed systems, the presented protocol can overcome the constraint on simultaneous failures by enabling every process to maintain the respective *RSN* of each same message assigned by every other live group member onto its own volatile memory. With this beneficial feature, it allows the system to survive a number of simultaneous failures except the whole system failure. Also, achieving the feature causes no extra control message by piggybacking the additional information on the control message for logging every previous protocol essentially needs. However, this feature may cause the time, which has elapsed until releasing the stalled messages, to be a little longer. But, the control message including the different *RSN*s of each message needs to be multi-casted to a group at once in our protocol whereas if the group size is *k*, the original SBML with *RSN* replication functionality requires that *k* individual control messages from its sender should be sent to all group members. The analysis and simulation results show that the proposed protocol can be a lightweight fault-tolerance technique for addressing the critical limitation of the original one based on group communication without *RSN* replication functionality, *i.e.*, capable of masking sequential failure only.

REFERENCES

1. L. Alvisi and K. Marzullo, "Message logging: Pessimistic, optimistic, causal, and optimal," *IEEE Transactions on Software Engineering*, Vol. 24, 1998, pp. 149-159.
2. G. R. Andrews, "Paradigms for process interaction in distributed programs," *ACM Computing Surveys*, Vol. 23, 1991, pp. 49-90.

3. R. Bagrodia, R. Meyer, M. Takai, Y. Chen, X. Zeng, J. Martin, and H. Y. Song, "Parsec: a parallel simulation environments for complex systems," *IEEE Computer*, Vol. 31, 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 *Proceedings of 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, 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 *Proceedings of 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, 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, 1985, pp. 63-75.
9. F. Cappello, A. Guermouche, and M. Snir, "On communication determinism in parallel HPC applications," in *Proceedings 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 *Proceedings of International Conference for High Performance Computing*, *Networking*, *Storage*, *and Analysis*, 2014, pp. 907-918.
11. E. Elnozahy, L. Alvisi, Y. Wang, and D. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys*, Vol. 34, 2002, pp 375-408.
12. D. Johnson and W. Zwaenpoel, "Sender-based message logging," in *Proceedings of the 7th International Symposium on Fault-Tolerant Computing*, 1987, pp. 14-19.
13. L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, Vol. 21, 1978, pp. 558-565.
14. 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.
15. H. F. Li, Z. Wei, and D. Goswami, "Quasi-atomic recovery for distributed agents," *Parallel Computing*, Vol. 32, 2009, pp. 733-758.
16. 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, 2011, pp. 127-139.
17. P. Jaggi and A. Singh, "Log based recovery with low overhead for large mobile computing systems," *Journal of Information Science and Engineering*, Vol. 29, 2013, pp. 969-984.

18. Y. Luo and D. Manivannan, "FINE: a fully informed and efficient communication-induced checkpointing protocol for distributed systems," *Journal of Parallel and Distributed Computing*, Vol. 69, 2009, pp. 153-167.
19. M. Powell and D. Presotto, "Publishing: a reliable broadcast communication mechanism," in *Proceedings of the 9th International Symposium on Operating System Principles*, 1983, pp. 100-109.
20. T. Ropars and C. Morin, "Active optimistic and distributed message logging for message-passing applications," *Concurrency and Computation: Practice and Experience*, Vol. 23, 2011, pp. 2167-2178.
21. 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, 1985, pp. 222-238.
22. R. E. Strom and S. A. Yemeni, "Optimistic recovery in distributed systems," *ACM Transactions on Computer Systems*, Vol. 3, 1985, pp. 204-226.
23. 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 *Proceedings of the 6th IEEE International Symposium on Cluster Computing and the Grid*, Vol. 2, 2006, p. 63.
24. T. T. Dinh, M. Lees, G. Theodoropoulos, and R. Minson, "Large scale distributed simulation of p2p networks," in *Proceedings of the 16th Euromicro Conference on Parallel*, *Distributed and Network-Based Processing*, 2008, pp. 499-507.
25. J. Xu, R. B. Netzer, and M. Mackey, "Sender-based message logging for reducing roll-back propagation," in *Proceedings of the 7th International Symposium on Parallel and Distributed Processing*, 1995, pp. 602-609.
26. B. Yao, K. Ssu, and W. Fuchs, "Message logging in mobile computing," in *Proceedings of the 29th International Symposium on Fault-Tolerant Computing*, 1999, pp. 14-19.



Jinho Ahn 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 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.