symmetry

MDPI

Article

Efficient Sender-Based Message Logging Tolerating Simultaneous Failures with Always No Rollback Property

Jinho Ahn

Division of AI Computer Science and Engineering, Kyonggi University, Suwon 16227, Republic of Korea; jhahn@kgu.ac.kr; Tel.: +82-31-249-9674

Abstract: Most of the existing sender-based message logging protocols cannot commonly handle simultaneous failures because, if both the sender and the receiver(s) of each message fail together, the receiver(s) cannot obtain the recovery information of the message. This unfortunate situation may happen due to their asymmetric logging behavior. This paper presents a novel sender-based message logging protocol for broadcast network based distributed systems to overcome the critical constraint of the previous ones with the following three features. First, when more than one process crashes at the same time, the protocol enables the system to ensure the always no rollback property by symmetrically replicating the recovery information at each process or group member connected on a network. Second, it can make the first feature persist even if the general form of communication for the system is a combination of point-to-point and group ones. Third, the communication overhead resulting from the replication can be highly lessened by making full use of the capability of the standard broadcast network in both communication modes. Experimental outcomes verify that, no matter which communication patterns are applied, it can reduce about 4.23∼9.96% of the total application execution time against the latest enabling the traditional ones to cope with simultaneous failures.

Keywords: distributed systems; concurrent failures; broadcast networks; message logging

Citation: Ahn, J. Efficient Sender-Based Message Logging Tolerating Simultaneous Failures with Always No Rollback Property. *Symmetry* **2023**, *15*, 816. https://doi.org/10.3390/sym15040816

Academic Editors: Kholod Ivan, Alexey Paznikov, Vasily Desnitsky, Silvio Pardi and Sergei D. Odintsov

Received: 6 February 2023
Revised: 13 March 2023
Accepted: 27 March 2023
Published: 28 March 2023

1. Introduction

A tightly coupled distributed system is well used in practice to provide a small or medium scale high-performance computing environment for missions'critical long-executing applications [1,2]. It is generally formed as a set of processes running on multiple hosts inter-connected on a broadcast network, called a cluster computing system. However, if at least one process or host crashes in the system, it may forget all the contents in its volatile storage, leading to making the system state inconsistent [3]. To counteract the negative effects of the misfortune, low overhead fault-tolerance techniques are essentially required for the system [4]. For this purpose, the rollback recovery technique is one of the appropriate tools and classified into two kinds, the checkpoint-based and message logging-based [5]. First, the checkpoint-based technique fulfills the fault-tolerance by letting each process periodically record its local state, called a checkpoint, on stable storage [6]. According to ensuring the system consistency, there are two types of checkpointing, as follows. Uncoordinated checkpointing does not require this sort of synchronization on checkpointing and, when process failures occur, finds a consistent global system state during recovery [7]. Due to this feature, a joint effort of every process made may become totally useless. Coordinated checkpointing forces all or related processes to have their local states consistent with each other's state on checkpointing, incurring higher failure-free overhead [8]. However, this technique has the insufficiency that, even if failed processes roll back to their latest consistent global checkpoint, they may not replay the messages in the same receipt order such as in their pre-failure states. This feature forces live processes to stop their execution and invalidate a part of the execution performed after their latest checkpoints.

Secondly, to overcome this kind of limitation, the message logging-based technique has been developed toward two directions, optimistic and pessimistic message logging [9]. The optimistic message logging technique [10,11] has the same disadvantage as the checkpointing-only recovery because it makes each process save the recovery information of each message only on the volatile storage of the receiver [12]. Pessimistic message logging protocols [13–19] let each message logged in secure places be unaffected by failures of message receivers, having no surviving process come back to its previous state. Despite their higher than normal operation cost, they become a popular fault-tolerance technique for the system.

Among them, sender-based message logging (hereafter noted as SBML) [13–18] is well known as one of the most lightweight choices because it uses only volatile logging at message senders compared with receiver-based ones [19] requiring stable storage accesses mandatorily. However, the traditional SBML protocols [13–18] cannot commonly handle simultaneous failures because, if the senders of each message fail together, its receiver(s) cannot obtain the recovery information of the message, called the determinant. This unfortunate situation may cause the system inconsistency. In this paper, a novel SBML protocol is presented to solve this important problem with the following features. First, the proposed protocol makes no rollback of the live processes permitted during recovery even in the case of concurrent process failures, called the always no rollback property [5], by the symmetric distribution of redundant determinants of each message. Second, it can still preserve the first feature even in the network environment, enabling a composite of point-to-point and group communication. Third, it can make full use of the performance efficiency the standard broadcast network generally embeds for decreasing the communication overhead occurring in the replication of the recovery information.

2. Preliminaries

2.1. Distributed System Model

We assume an asynchronous distributed system with no global memory, consisting of a set of processors, processes and communication channels. Processes can communicate with each other by exchanging messages at finite but arbitrary transmission delays through unreliable channels, meaning messages may be dropped or duplicated on these channels. However, it is assumed that the channels are immune to partitioning. In addition, we assume that processes may fail based on the crash-failure model, in which they lose contents in their volatile memories and stop their executions [5]. This system is augmented with an unreliable failure detector [20] in order to solve the impossibility problem on distributed consensus. A process p starting at its initial state, s_0^p, generates a sequence of events, [e_1^p, e_2^p, …, e_k^p]. These events are classified into internal, sending, and delivery. Process p may produce an internal event to execute a particular computation without any communication. A sending event of a message m, denoted by sending(m), is generated by sending message m to another process, and a delivery event of m, delivery(m), by actually delivering the message after its receipt to its corresponding application. After applying the sequence of events [e_1^p, e_2^p, …, e_k^p] to s_0^p in order, p has a unique local state $s_k^p (k > 0)$. Events of processes occurring in a failure-free execution are ordered using Lamport's, and happened before the relationship [5].

2.2. Related Works

The coordinated or uncoordinated checkpoint-based recovery with optimistic message logging may force live processes to nullify a part of their execution due to process failures [12]. The primary reason is that, if they crash, processes lose the log information of received messages in their volatile memories. In this case, if the surviving processes have received any messages from those failed, depending on the lost ones, their states may be inconsistent with the states of the recovered even after completing its recovery procedure. Let us take a look at the shortcomings with an illustration. In Figure 1, process p_x receives two messages m_1 and m_2 from two processes, s_α and s_β, in order, individually. Hereafter, it transmits a message m_3 to another process p_y. If p_x crashes later and tries to recover, like in

this figure, it should come back to its latest checkpoint Ck_x^j and replay the receipt of the two messages. However, as p_x cannot obtain their receive sequence numbers(hereafter noted as *rsn*s) from anywhere, m_2 may arrive at p_x earlier than m_1 during recovery, unlike in the pre-failure state. If so, it creates a message m_z to p_y different from m_3 that p_x generated before the failure. To make the system have a consistent state, this situation causes a live process p_y to return to its most recent checkpoint Ck_y^l before delivering m_3 and then receiving m_z. The rollback recovery of p_y may also force other live processes, although not demonstrated in this figure, to revert to their past states in the series. To minimize this kind of invalidation effect of normal operation execution, pessimistic message logging is an essential building block that enables each failed process to carry out its own recovery procedure consistently without any rollback of live ones, which is called the always no rollback property [5].

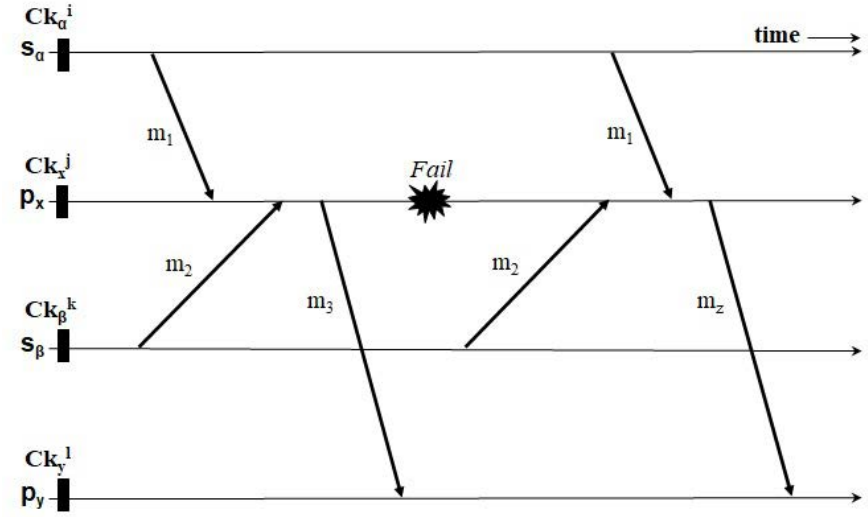

Figure 1. Example of replaying messages when p_x fails.

SBML is one of the pessimistic logging techniques holding this property on a sequential failure tolerance only: each message is logged in the volatile storage of its sender [13–18]. This asymmetric behavior allows each crashed process to attain the determinant and the contents of every message it received before the failure from the sender. After the process replayed the message at the same receipt position, its state always becomes consistent with the current one of an arbitrary live process. For example, in Figure 1, after performing SBML during normal operation, s_α and s_β currently possess the recovery information of m_1 and m_2 in their respective memories. When it crashes, p_x can collect the determinants and message contents of the two messages from them and replay the delivery of the messages in the same *rsn* order it was in before the failure, regenerating the identical message m_3 sent to p_y. Then, the final recovered state of p_x becomes consistent with the current one of p_y.

However, the previous SBML protocols [13–18] commonly possess the critical constraint due to its asymmetric logging behavior: It can only tolerate consecutive failures, not simultaneous failures. In other words, each crashed process cannot replay messages received in the same *rsn* order like before the failure if their senders fail together, because they lose the determinants of the messages in their volatile memories. This consequence may give rise to another type of system inconsistency. In particular, when a sender transmits a message m to a group of processes on the network, the protocols have no functionality to handle the issue of the existence of multiple *rsn*s of m assigned by the group of processes. Moreover, it possess no capability of decreasing the communication cost incurred by the replication by utilizing the embedded efficiency of the broadcasting network.

3. The New SBML Protocol

3.1. Basic Concepts and Algorithms

A new SBML protocol is designed with the following advantageous features for a distributed system composed of a set of processes on a broadcast network.

- None of the live processes in the system are rolled back during recovery, even in the case of concurrent process failures.
- The protocol supports a fault-tolerance for distributed applications exchanging messages in a mix of point-to-point and group communication modes.
- With little communication cost, it can maintain the recovery information of each application message on redundant volatile storages in a symmetric manner.

To fulfill all the three requirements in the proposed protocol, each process p_i should keep the following variables for saving a replicated recovery information of messages.

- ssn_i: the sequence number of the most recent one among all the messages p_i has transmitted since its initial execution state.
- rsn_i: the sequence number of the most recent one among all the messages p_i has delivered to applications since its initial execution state.
- $nSLog_i$: a set keeping the recovery information of each message p_i transmitted. Its element e is composed of the identifier of the receiver (*rid*), send sequence number (*ssn*), list of the receive sequence numbers (*rsnList*), and data (*data*) of the message. Here, the first field *rid* can be the identifer of a process or a process group. Moreover, the third field *rsnList* is a set whose component is a form of (*pid*, *rsn*) of the message p_{pid} received and assigned to *rsn*. It may contain multiple *rsn*s of each sent message if the message is transferred to a group of processes.
- $nRLog_i$: a set keeping the determinant of each message p_i received. Its element e is composed of the identifier of the sender (*sid*), *rid*, *ssn*, and *rsnList* of the message. Here, the second field *rid* and the fourth field *rsnList* have the same respective meanings as in $nSLog_i$.
- $ssnTable_i$: a table for sensing duplication of application messages already delivered that their senders have regenerated in their recovery procedures. Its field $ssnTable_i[j]$ contains the *ssn* of the most recent one that p_i has received from another process p_j.

The protocol consists of five primary procedures for p_i to make the determinant of each received message m, maintained not only at its sender's and receiver's volatile storages, $nSLog_{m's\ sender}$ and $nRLog_i$, but also at those of others on the network, $nRLog_j$($j \neq i$ and $j \neq m$'s sender). The first procedure, SEND_M(), allows a sender to transmit each message and keep the recovery information for the message in its volatile log. The second, RECEIVE_M(), has each process receiving the message send its *rsn* to the group of the process as well as its sender, keep the determinant of the message in its volatile log and indicate the execution of all the send operations invoked after the message should be delayed. The third, RECEIVE_DET(), enables the sender or the group member to insert the *rsn* into the element for recovering the message and then notifies the receiver of the completion of the volatile logging task. The fourth, RECEIVE_ACK(), checks whether the receiver obtains the notification from both the sender and every other group member. If so, it actually performs all the send operations delayed between the message and its immediate next message. The last, TAKE_CHECKPOINT(), forces each process to save its checkpointed state with its local variables on the stable storage and then performs all the send operations delayed before the checkpoint. With this symmetric feature, it can hold the always no orphan property, which the receiver-based pessimistic message logging using stable storage must mandatorily have for tolerating simultaneous crashes.

Let us examine how its logging procedure works according to communication types in detail. First, when a process receives an application message in the point-to-point communication mode, it allows the other processes to maintain the duplicate determinants of the message including the *rsn* of the message on their volatile memories. Second, every process receiving a message destined to its group g should hold determinants recording all

the distinct *rsn*s assigned by the other group members as well as itself in its own storage in a cooperative manner. Moreover, the new protocol is developed to make better use of the performance efficiency that the standard broadcast network generally embeds in a wise manner, minimizing the communication overhead coming from the replication of the recovery information for overcoming multiple failures occurring at the same time. Figure 2 formally describes all the procedures in the proposed protocol that each process performs depending on the type of event action.

Procedure SEND_M(rid, $data$)
  increment ssn_i **by** one ;
  if(rid is a group identifier) **then**
    broadcast $m(i, rid, ssn_i, data)$ **to** group rid ;
  else
    send $m(i, rid, ssn_i, data)$ **to** p_{rid} ;
  $nSLog_i \leftarrow nSLog_i \cup \{(rid, ssn_i, \{\}, data)\}$;

Procedure RECEIVE_M($m(sid, rid, ssn, data)$) from p_j
  if(($m.rid = i$) $\vee$ (p_i is a member of group $m.rid$ except m's sender)) **then**
    if($ssnTable_i[m.sid] < m.ssn$) **then**
      increment rsn_i **by** one ;
      $ssnTable_i[m.sid] \leftarrow m.ssn$;
      broadcast $det(m.sid, m.ssn, m.rid, i, rsn_i)$ **to** p_j and the process group of p_i ;
      $nRLog_i \leftarrow nRLog_i \cup \{(m.sid, m.rid, m.ssn, \{(i, rsn_i)\})\}$;
      delay execution of all the send operations depending on m ;
      deliver $m.data$ **to** its corresponding application ;
    else
      retrieve $\exists\, e \in nRLog_i$ **st** $((e.sid = m.sid) \wedge (e.ssn = m.ssn))$;
      find $\exists\, l \in e.rsnList$ **st** $(l.pid = i)$;
      send $det(e.sid, e.ssn, e.rid, l.pid, l.rsn)$ **to** p_j ;

Procedure RECEIVE_DET($det(sid, ssn, rid, pid, rsn)$) from p_j
  if($det.sid = i$) **then**
    find $\exists\, e \in nSLog_i$ **st** $(e.ssn = det.ssn)$;
    $e.rsnList \leftarrow e.rsnList \cup \{(det.pid, det.rsn)\}$;
  else if($det.pid \neq i$) **then**
    find $\exists\, e \in nRLog_i$ **st** $((e.sid = det.sid) \wedge (e.ssn = det.ssn))$;
    if(e exists) **then**
      $e.rsnList \leftarrow e.rsnList \cup \{(det.pid, det.rsn)\}$;
    else
      $nRLog_i \leftarrow nRLog_i \cup \{(det.sid, det.rid, det.ssn, \{(det.pid, det.rsn)\})\}$;
  send $ack(i, det.ssn, det.rid, det.pid, det.rsn)$ **to** p_j ;

Procedure RECEIVE_ACK($ack(sid, ssn, rid, pid, rsn)$) from p_j
  if($ack.pid = i$) **then**
    if(ack for the message with ($ack.sid$, $ack.ssn$, $ack.rid$, $ack.pid$, $ack.rsn$) is received
      from both $ack.sid$ and every other live member) **then**
      execute all the send operations delayed before the message
        whose rsn value is ($ack.rsn$+1) ;

Procedure TAKE_CHECKPOINT()
  take its local checkpoint with (rsn_i, ssn_i, $nSLog_i$, $nRLog_i$, $ssnTable_i$)
    on the stable storage ;
  execute all the send operations delayed before this checkpoint ;

Figure 2. Procedures of each process p_i for the proposed protocol.

We show pictorial examples to help understand how the protocol can hold all the beneficial features stated above. In Figures 3–6, it is assumed four processes s_α, p_x, p_y, p_z can communicate with each other through a broadcast network. In these examples, s_α first sends an application message m_1 to a process group g composed of p_x, p_y and p_z in Figures 3 and 4, and then a point-to-point one m_2 to p_x in Figures 5 and 6. Suppose the initial values of ssn_α, rsn_x, rsn_y and rsn_z in Figure 3 are d, a, b and c. In this figure, s_α increments ssn_α by one and transmits m_1 with ssn_α to the broadcast network (by invoking procedure SEND_M() in Figure 2). After that, it creates an element for recording the recovery information of m_1 and saves both the $tag(m_1)$ and data of m_1 on $nSLog_\alpha$. Here, $tag(m_1)$ consists of *sid*, *rid*, and *ssn* of m_1. At this time, as the *rsn*s of m_1 are not decided by its receivers, the list type field recording some pairs of the identifier of the respective receiver and the *rsn* of m_1 assigned by the receiver is initialized to an empty set. As m_1 is destined to group g, each member can sense its presence as group message and obtain it from the network. Hereafter, it increments its own *rsn* by one and sends a control message *det* containing a pair of its identifier and the *rsn* into the network while creating and saving the determinant of m_1, including the pair on its volatile memory (by invoking procedure RECEIVE_M() in Figure 2). The control message is transferred not only to the sender of m_1, but also to every other member.

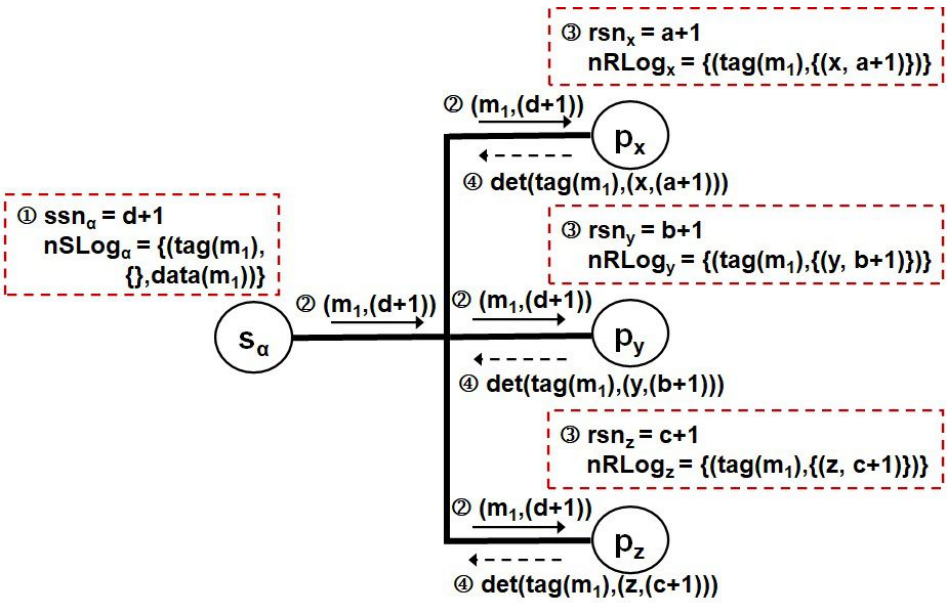

Figure 3. Example of transmitting a group message m_1 to p_x, p_y and p_z in the proposed SBML.

Then, in Figure 4, the sender s_α extracts the pair from the control message and inserts it into the *rsn* list field of the corresponding element in $nSLog_\alpha$ (by invoking procedure RECEIVE_DET() in Figure 2). In this case, as the *rsn*s of p_x, p_y, and p_z may be different from each other, the three pairs should be kept on the list field. Similarly, when perceiving the group control messages from the others, each member maintains on its volatile log all other pairs including the *rsn*s of m_1 assigned by the others (by invoking procedure RECEIVE_DET() in Figure 2). Afterward, it transmits another control message $ack(tag(m_1))$ to group g. After receiving the second control message from both the sender and the other members, each member can recognize the procedure for replicating the *rsn* of m_1, which, assigned by itself, has been terminated (by invoking procedure RECEIVE_ACK() in Figure 2).

Figures 5 and 6 show an example of sending a unicast message m_2 to p_x. After increasing its *ssn*, s_α lets m_2 with the *ssn* transmitted out to p_x, and then puts m_2's initial recovery information in its volatile storage in Figure 5 (by invoking procedure SEND_M() in Figure 2). In this case, as the only target of m_2 is p_x, p_x broadcasts a control message *det* with

its incremented *rsn* to the network (by invoking procedure RECEIVE_M() in Figure 2). It also makes and keeps m_2's determinant in $nRLog_x$. Detecting the control message from the network, s_α, p_y and p_z pull out a pair of p_x's identifier and the *rsn* of m_2 from the message, and then update their respective volatile logs with the pair like in Figure 6 (by invoking procedure RECEIVE_DET() in Figure 2). Next, they acknowledge their own volatile logging completion by conveying $ack(tag(m_1))$ to p_x (by invoking procedure RECEIVE_ACK() in Figure 2). Therefore, the protocol can reduce the number of the first control message down to one required to manage the recovery information redundancy for each application by making full use of the broadcasting capability in both communication modes.

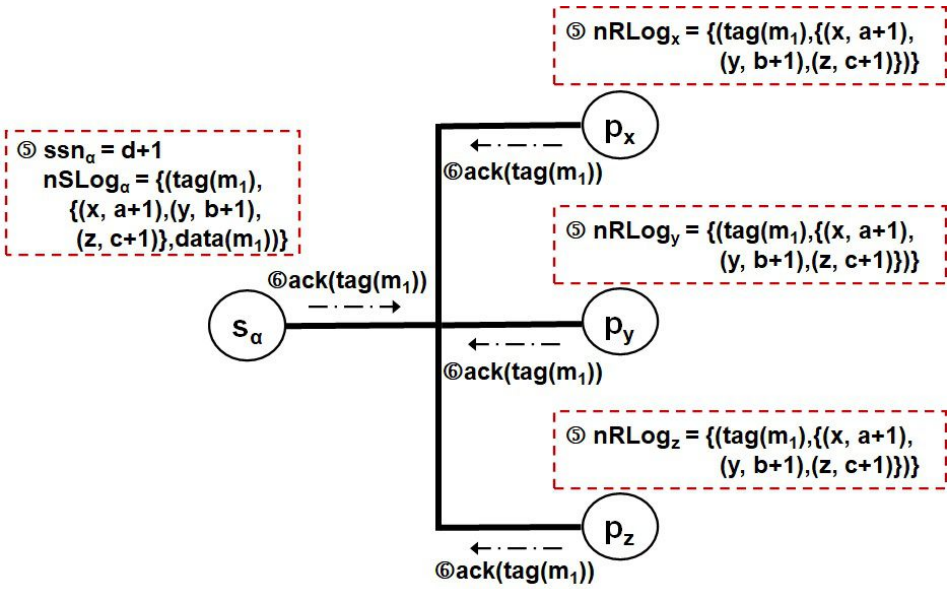

Figure 4. Example of replicating the three distinct determinants of m_1 on all the volatile storages of s_α, p_x, p_y, and p_z together in the proposed SBML.

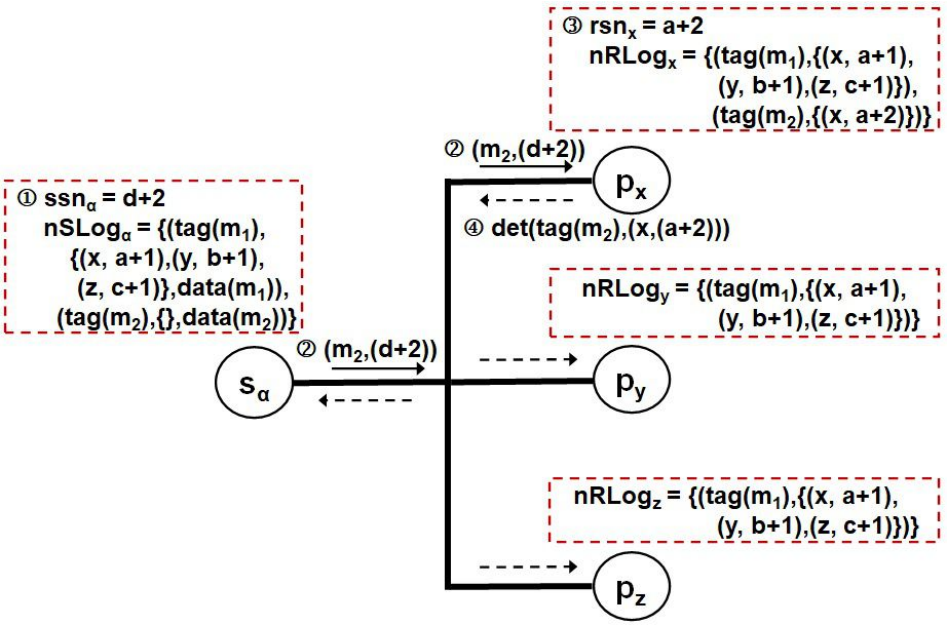

Figure 5. Example of transmitting a point-to-point message m_2 to p_x in the proposed SBML.

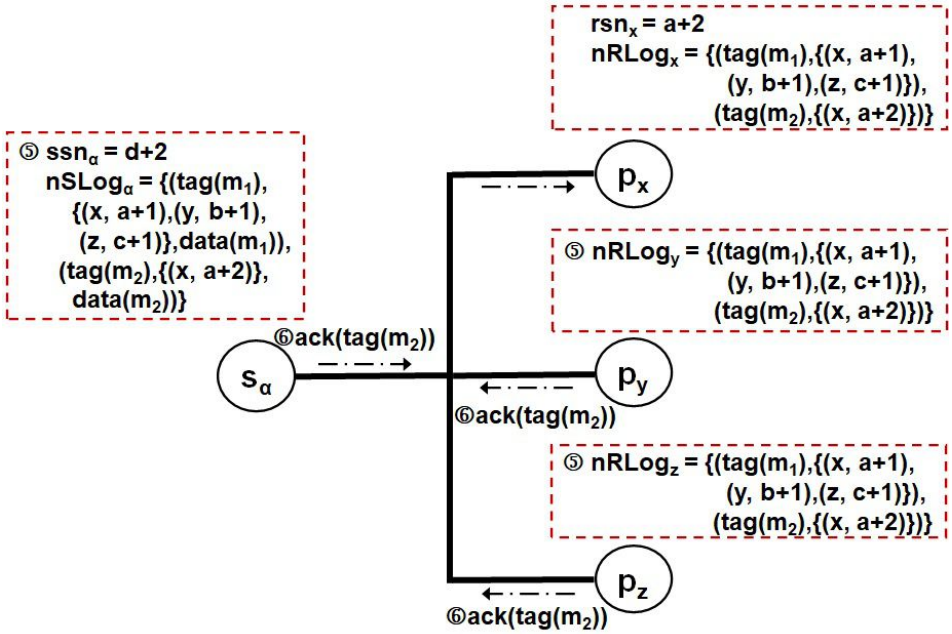

Figure 6. Example of replicating the determinant of m_2 on all the volatile storages of s_α, p_y, and p_z in the proposed SBML.

Let us verify whether the protocol with the symmetric behavior stated earlier can satisfy the first two requirements using a recovery scenario. Figure 7 illustrates the situation that all processes on the network except p_z crash from the state of Figure 6. Each failed process rolls back to its latest checkpoint and disseminates a recovery message into the broadcast network. At that time, a live process p_z possesses in $nRLog_z$ the determinant(s) of every point-to-point or group message transmitted on the network during normal operation. Perceiving the recovery request from any crashed process, it provides them for the failed one using just one message transfer, such as in Figure 8. Hereafter, every recovering process can be restored into a pre-failure state consistent with the current state of p_z.

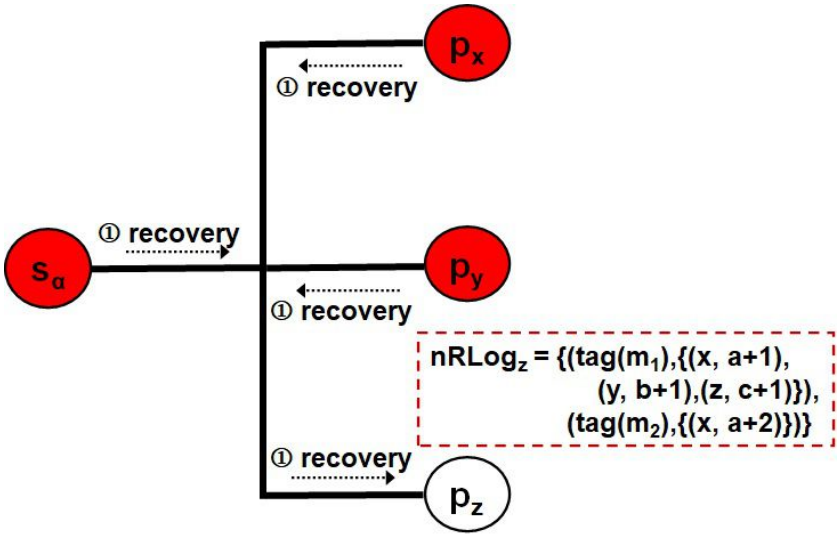

Figure 7. Example of the three failed processes requesting their recovery information by broadcasting to the network in the proposed SBML.

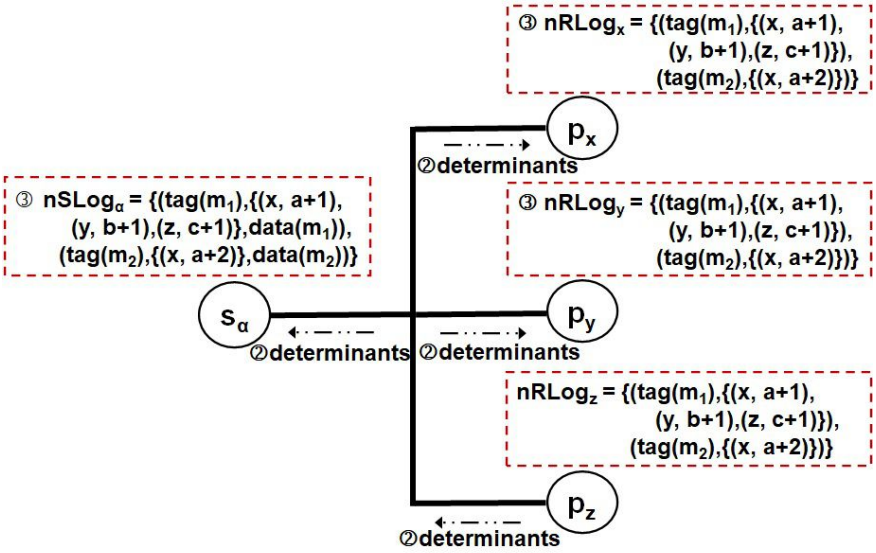

Figure 8. Example of enabling p_z to provide the failed processes with all the determinants for restoring their respective pre-failure states in the proposed SBML.

3.2. Correctness

In this section, we prove that the proposed protocol guarantees a consistent recovery and satisfies the always no rollback property even in the case of simultaneous failures by Theorems 1 and 2, respectively. In particular, Theorem 2 means even though more than one process fails at the same time, they have only to roll back to their latest checkpointed states, while all other live processes can continue their own executions without requiring their rollbacks. The proof of the second theorem verifies that the number of live processes rolled back by process failures of our protocol is 0 at all times.

Theorem 1. *Even if the concurrent failures of processes on a broadcast network occur, our proposed protocol always enables the system to be recovered consistently.*

Proof. Let us denote the set of all processes on a broadcast network by N ($3 \leq |N|$), the set of all concurrent crashed processes $\in N$, by $CRASHED$, and the set of all live processes $\in N$, by $SURVIVING$. We prove the correctness of this theorem by induction on the number of all the concurrent crashed processes, $|CRASHED|$ ($2 \leq |CRASHED| \leq |N|$).
[Base case]:
It is supposed $|CRASHED| = 2$, i.e., the two processes p_x and p_y, crash simultaneously. In this case, there exists at least one live process as $|SURVIVING| = |N| - |CRASHED|$. Before the failures, the protocol makes every process $p_z \in SURVIVING$ obtain from both p_x and p_y and maintain in $nSLog_z$ or $nRLog_z$ the determinants of all the point-to-point or group messages they received since their respective latest checkpoints by executing procedure RECEIVE_DET() in Figure 2. Therefore, after each failed process $\in CRASHED$ has obtained them from p_z and replayed their corresponding messages in the same order before the crashes, it can be restored to be a state consistent with p_z's current state.
[Induction hypothesis]:
Suppose that the theorem is true in the case that $|CRASHED| = k (2 \leq k \leq |N| - 1)$.
[Inductive step]:
By induction hypothesis, the recovered state of each process $\in CRASHED$ is consistent with the current states of all the processes $\in SURVIVING$ despite the occurrence of k concurrent failures. With this assumption, let us verify whether the theorem is true in the case $|CRASHED| = k + 1$. Suppose a process $p_w \in SURVIVING$ joins $CRASHED$ at the same time interval. There are two cases we should consider.
Case 1: $(k + 1) < |N|$.

In this case, the subsequent steps are similar to the base case mentioned earlier.
Case 2: $(k + 1) = |N|$.
In this case, there exists no live process in the system, i.e., $|SURVIVING| = 0$. It means the recovered state of the system always becomes consistent even though every failed process replays all the messages it delivered before the crashes in any receipt order.
By the induction, the protocol makes the system have a consistent state despite any number of simultaneous process crashes on a broadcast network. □

Theorem 2. *Our proposed protocol ensures the always no rollback property even if multiple processes crash at the same time.*

Proof. Let us prove this theorem by contradiction. It is assumed the protocol may not ensure the always no rollback property in case of simultaneous failures. This assumption means that there is at least one failed process p making live processes roll back for the system consistency. Suppose m is an arbitrary message p has received and completely logged since its latest checkpoint according to the protocol. Two cases should be considered.
Case 1: m is a group message.
Case 1.1: process q is the sender of m and does not crash.
As q maintains in $nSLog_q$ m's rsns all group members, including the p assigned for m before p's failure; p can directly obtain them from q during recovery. Thanks to this procedure, p can always replay m in its pre-failure receipt order and then regenerate every point-to-point or group message which p sent to live process(es) between m and its immediate next message received before the failure. Therefore, p's state becomes consistent with those of the live process(es).
Case 1.2: process q is the sender of m and crashes.
Two sub-cases should be considered.
Case 1.2.1: every process in the group fails.
There exists no live process affected by the recovered state of p regardless of m's replaying order. Therefore, the system has no inconsistency problem.
Case 1.2.2: at least one live process r in the group exists.
The completion of logging m in the protocol has all live group members including r obtain the rsns they assigned for m together. Thanks to its behavior, r can provide p with the rsns in $nRLog_r$. Therefore, p can always replay m in its pre-failure receipt order and then regenerate every point-to-point or group message which p sent to live process(es) between m and its immediate next message received before the failure. Therefore, p's state becomes consistent with r's state.
Case 2: m is a point-to-point message.
All the subsequent steps are similar to case 1.
Therefore, the protocol makes no live process roll back even if multiple processes fail concurrently. This contradicts the hypothesis. □

4. Performance Evaluation

4.1. Simulation Environments

In this section, an extensive simulation has been conducted to measure the performance of the proposed protocol *OURS* against the latest counterpart, called *ORIGINAL-R*, with PARSEC simulation language [21]. As mentioned in Section 2.2, all the existing SBML protocols [13–18] have no functionality to overcome concurrent failures due to their asymmetric logging behavior. For this reason, they cannot be compared with *OURS* in terms of failure-free overhead incurred by the creation, delivery and management of the redundant recovery information for this purpose. To verify how much performance benefit *OURS* can provide, we developed an enhanced replication-enabling SBML protocol *ORIGINAL-R* based on the common structure of all the previous SBML ones to cope with simultaneous failures for comparative study. In *ORIGINAL-R*, whenever each process obtains a point-to-point or group communication type of application message m from another, the receiver

transmits to every other process on a network an individual control message containing the determinant of m. Moreover, even if the process receiving the control message is not m's sender, $ORIGINAL\text{-}R$ forces the process to keep the determinant of m in its volatile memory and convey an acknowledgement of the completion of logging m to the receiver of m.

A simulated distributed system is composed of n hosts exchanging point-to-point or group messages transferred on a broadcast network with 100 Mbps transmission capacity and 1 ms propagation delay. All processes in the system start and finish their corresponding executions together. Each process running on one host can send an application message whose size is scaled from 1 KB to 1 MB to an individual process or a group of processes, following an exponential distribution with a mean value of 3 sec. If one process performs one message send operation to a group, the network is capable of conveying the message to every group member using an IP multicast functionality [22]. The capacity of the volatile storage for the logging per process is 256 MB. The interval between consecutive checkpoints taken at a process conforms to an exponential distribution with a mean value of 5 min.

To verify our claim mentioned earlier, the two protocols make a comparison of efficiency with a key performance element ($T_{execution}$) in terms of failure-free overhead resulting from their logging procedures for getting over concurrent failures. $T_{execution}$ means the total time(marked in minutes) required to complete the execution of a distributed application. Moreover, to perform a rich analysis of logging overhead evaluation, four types of distributed applications, serial, circular, hierarchical and irregular, have been executed depending on the communication pattern [23]. This measurement method is a great help to identify a more accurate performance change arising from the susceptibility of communication patterns. We took an average of the experimental results of multiple runs.

4.2. Comparison Results

Figures 9–12 illustrate $T_{execution}$ of the two protocols by changing the size of the process group on a broadcast network, $Group_{size}$, ranging from 6 to 16 according to inter-process communication patterns. When $Group_{size}$ increases, the values of $T_{execution}$ for both proportionally become higher regardless of four communication patterns in Figures 9–12. The results are determined because two protocols both generate quite a few additional control messages for carrying the determinant of each application message to every group member in order to prepare for simultaneous failures. However, these figures show that $T_{execution}$ of $ORIGINAL\text{-}R$ is bigger than that of *Ours* and the difference between the two arises quickly in proportion to the growth of $Group_{size}$, signifying that *Ours* outperforms the counterpart, especially in large-scale broadcast networked systems. The phenomenon equally occurs in any communication pattern, such as in Figures 9–12. The reduction rate of $T_{execution}$ of *Ours* over $ORIGINAL\text{-}R$ is scaled to 4.64∼9.96% in a serial pattern in Figure 9, 4.24∼9.52% in a circular pattern in Figure 10, 4.66∼9.19% in a hierarchical pattern in Figure 11, and 4.62∼9.49% in an irregular pattern in Figure 12. The primary cause is that, unlike $ORIGINAL\text{-}R$, *Ours* can transmit each control message to every other process or member with only one broadcast transfer of it for conveying the determinants or acknowledgements of the completion of volatile logging. This effectiveness may be enlarged in a group communication mode because the number of message receivers greatly increases.

With these results, we can recognize that when a distributed application is executed combining point-to-point and group messages, *Ours* can achieve the goal of surviving k concurrent failures with much less communication overhead coming from the symmetric distribution of redundant determinants to each process or member compared with $ORIGINAL\text{-}R$, by making better use of the cost-effective characteristic of broadcast networks irrespective of their delivery pattern.

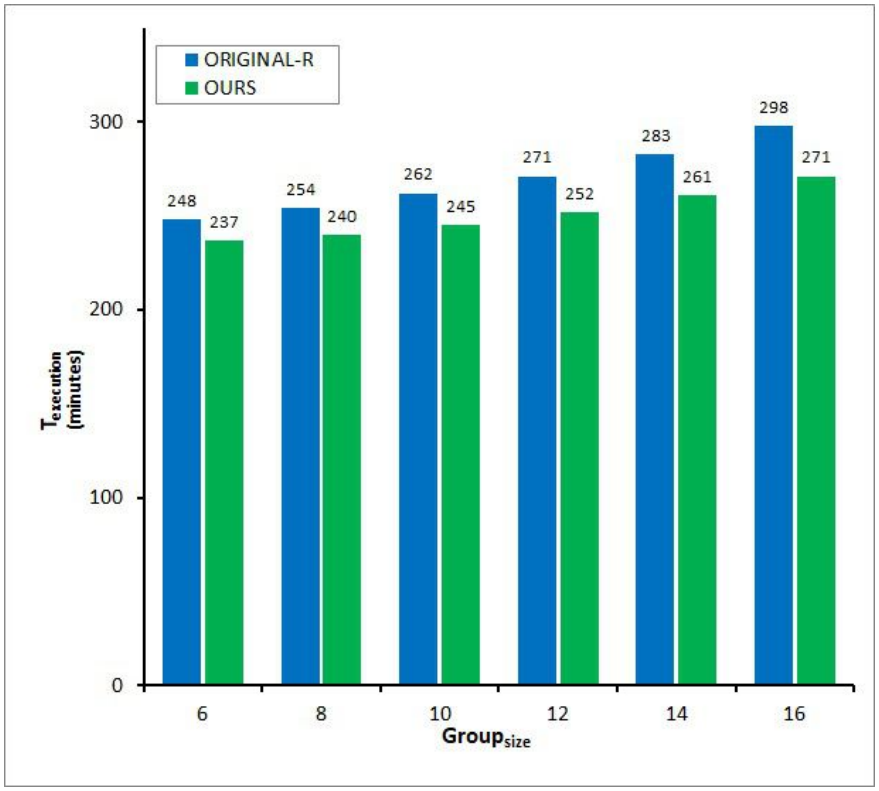

Figure 9. $T_{execution}$ in case of serial pattern.

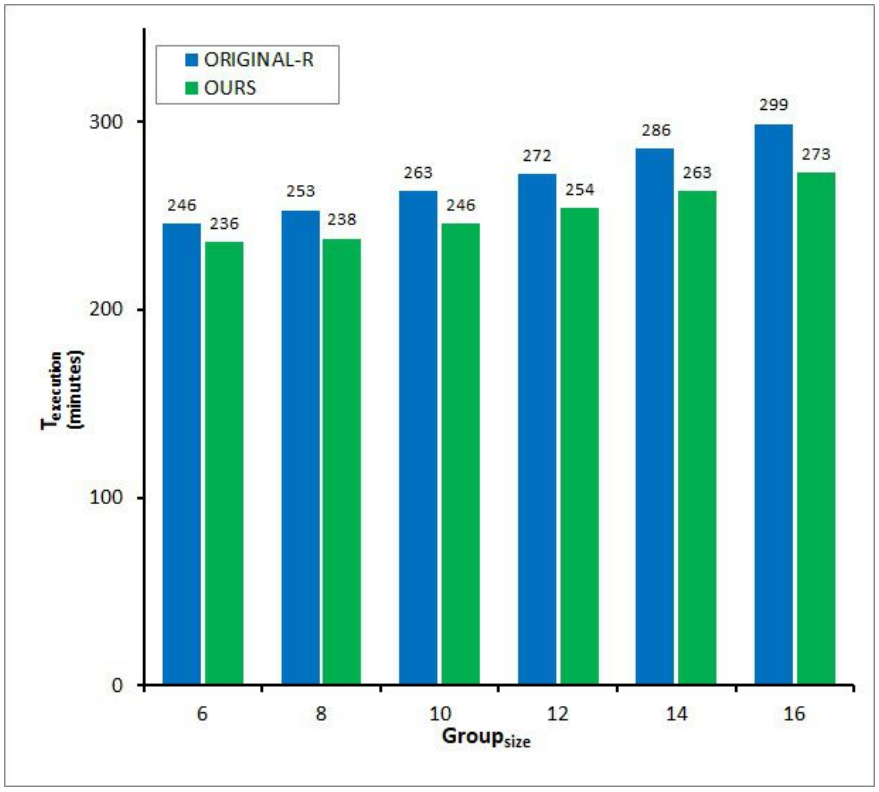

Figure 10. $T_{execution}$ in case of circular pattern.

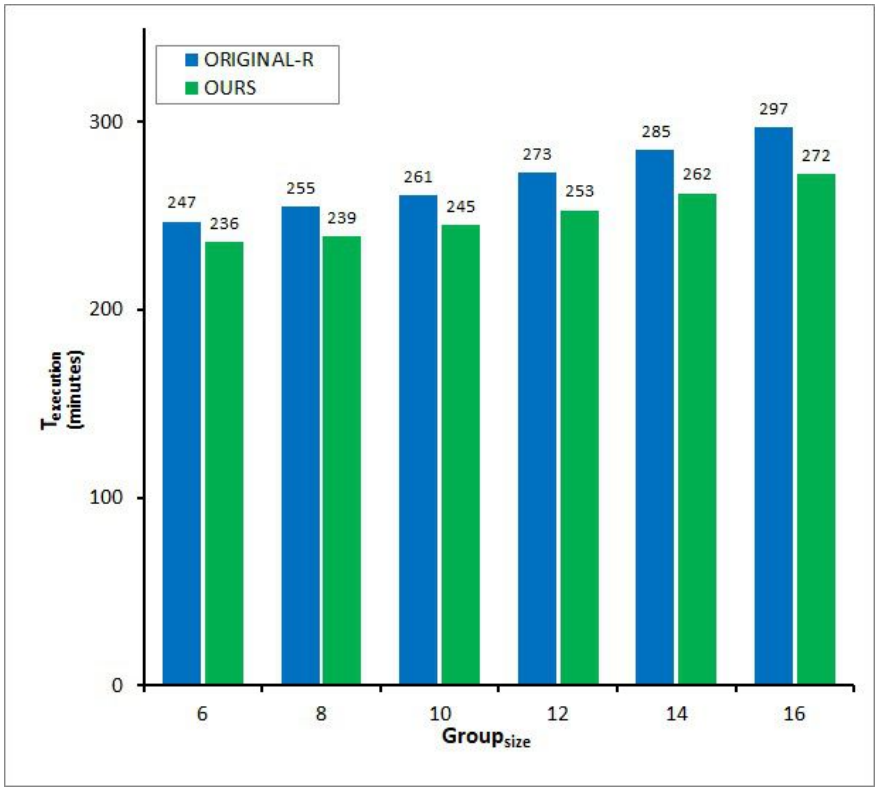

Figure 11. $T_{execution}$ in case of hierarchical pattern.

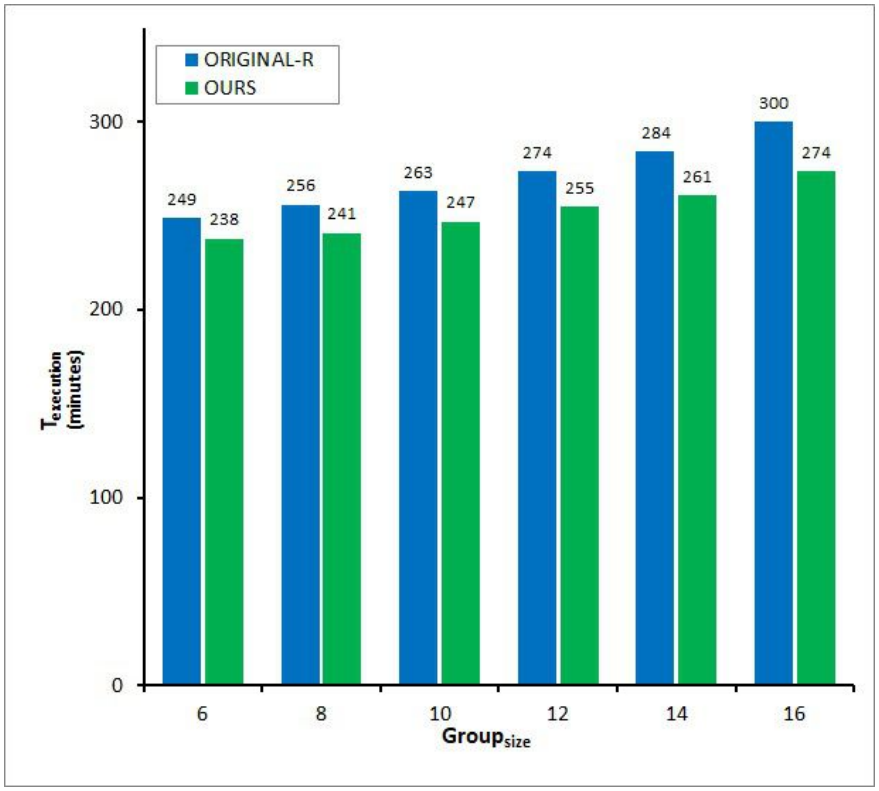

Figure 12. $T_{execution}$ in case of irregular pattern.

5. Conclusions

Unlike the previous SBML protocols, the *OURS* presented in this paper possesses the three desirable features, as follows. First, when more than one process crashes at the same

time, *OURS* allows a distributed system to make no rollback of the live processes permitted during recovery, even in the case of simultaneous failures by symmetrically replicating recovery information at each process or group member connected on a network. Second, *OURS* can make the first feature persist even if the general form of communication for the system is a combination of point-to-point and group ones. Third, the communication overhead resulting from the replication can be highly lessened by making full use of the capability of the standard broadcast network in both communication modes. Through Theorem 2 in Section 3.2, we verified that *OURS* prevents any surviving process from rolling back even in the case of concurrent failures, i.e., ensuring the always no rollback property. Moreover, from the experimental outcomes, we recognized that, no matter which communication patterns are applied, *OURS* can reduce about 4.23∼9.96% of the total application execution time against the latest, enabling the traditional ones to cope with simultaneous failures. In particular, this enhancement may be magnified in the group communication mode that may significantly grow the number of receivers of messages generated.

For future works, we attempt to access the practicality of our protocol *OURS* by applying it into Open MPI library based high performance systems employing a large number of computing nodes with no message logging capabilities [24]. In particular, the protocol will be implemented with a representative failure detection mechanism combining a periodic liveness check with sporadic check.

Funding: This research was funded by Kyonggi University, grant number 2020-032.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data that support the findings of this study are available from the corresponding author, J.A., upon reasonable request.

Conflicts of Interest: The author declares no conflict of interest.

References

1. Wang, X.; Li, H.; Sun, Q.; Guo, C.; Zhao, H.; Wu, X.; Wang, A. The g-Good-Neighbor Conditional Diagnosability of Exchanged Crossed Cube under the MM* Model. *Symmetry* **2022**, *14*, 2376. [CrossRef]
2. Wang, S.; Yao, Y.; Zhu, F.; Tang, W.; Xiao, Y. A Probabilistic Prediction Approach for Memory Resource of Complex System Simulation in Cloud Computing Environment. *Symmetry* **2020**, *12*, 1826. [CrossRef]
3. Mansouri, H.; Pathan, A. Checkpointing distributed computing systems: An optimisation approach. *Int. J. High Perform. Comput. Appl.* **2019**, *15*, 202–209. [CrossRef]
4. Chlebus, B.S.; Kowalski, D.R.; Olkowski, J. Brief announcement: Deterministic consensus and checkpointing with crashes: Time and communication efficiency. In Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing, Salerno, Italy, 25–29 July 2022; pp. 106–108.
5. Elnozahy, E.; Alvisi, L.; Wang, Y.; Johnson, D. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.* **2002**, *34*, 375–408. [CrossRef]
6. Lion, R.; Thibault, S. From tasks graphs to asynchronous distributed checkpointing with local restart. In Proceedings of the IEEE/ACM 10th Workshop on Fault Tolerance for HPC at eXtreme Scale, Atlanta, GA, USA, 11 November 2020; pp. 31–40.
7. Jayasekara, S.; Karunasekera, S.; Harwood, A. Optimizing checkpoint-based fault-tolerance in distributed stream processing systems: Theory to practice. *Softw. Pract. Exp.* **2022**, *52*, 296–315. [CrossRef]
8. Abdelhafidi Z.; Djoudi M.; Lagraa, N.; Yagoubi, M.B. FNB: Fast non-blocking coordinated checkpointing protocol for distributed systems. *Theory Comput. Syst.* **2015**, *57*, 397–425. [CrossRef]
9. Meyer, H.; Rexachs, D.; Luque, E. Hybrid message pessimistic logging. improving current pessimistic message logging protocols. *J. Parallel Distrib. Comput.* **2017**, *104*, 206–222. [CrossRef]
10. Ropars, T.; Morin, C. Active optimistic and distributed message logging for message-passing applications. *Concurr. Comput. Pract. Exp.* **2011**, *23*, 2167–2178. [CrossRef]
11. Ropars, T.; Morin, C. Improving message logging protocols scalability through distributed event logging. In Proceedings of the 16th International Euro-Par Conference, Ischia, Italy, 31 August–3 September 2010; pp. 511–522.
12. Bouteiller, A.; Ropars, T.; Bosilca, G.; Morin, C.; Dongarra, J. Reasons for a pessimistic or optimistic message logging protocol in MPI uncoordinated failure, recovery. In Proceedings of the 2009 IEEE International Conference on Cluster Computing and Workshops, New Orleans, LA, USA, 31 August–4 September 2009; pp. 1–9.

13. Ahn, J. Enhanced sender-based message logging for reducing forced checkpointing overhead in distributed systems. *IEICE Trans. Inf. Syst.* **2021**, *E104-D*, 1500–1505. [CrossRef]
14. Ahn, J. Scalable sender-based message logging protocol with little communication overhead for distributed systems. *Parallel Process. Lett.* **2019**, *29*, 1–10. [CrossRef]
15. Johnson, D.; Zwaenpoel, W. Sender-based message logging. In Proceedings of the 7th International Symposium on Fault-Tolerant Computing, Pittsburgh, PA, USA, 6–8 July 1987; pp. 14–19.
16. Gupta, B.; Nikolaev, R.; Chirra, R. A recovery scheme for cluster federations using sender-based message logging. *J. Comput. Inf. Technol.* **2011**, *19*, 127–139. [CrossRef]
17. Jaggi, P.; Singh, A. Log based recovery with low overhead for large mobile computing systems. *J. Inf. Sci. Eng.* **2013**, *29*, 969–984.
18. Luo, Y.; Manivannan, D. HOPE: A hybrid optimistic checkpointing and selective pessimistic mEssage logging protocol for large scale distributed systems. *Future Gener. Comput. Syst.* **2012**, *28*, 1217–1235. [CrossRef]
19. Kumari, P.; Kaur, P. Checkpointing algorithms for fault-tolerant execution of large-scale distributed applications in cloud. *Wirel. Pers. Commun.* **2021**, *117*, 1853–1877. [CrossRef]
20. Chandra, T.D.; Toueg, S. Unreliable failure detectors for reliable distributed systems. *J. ACM* **1996**, *43*, 225–267. [CrossRef]
21. Bagrodia, R.; Meyer, R.; Takai, M.; Chen, Y.; Zeng, X.; Martin, J.; Song, H.Y. Parsec: A parallel simulation environments for complex systems. *Comput. J.* **1998**, *31*, 77–85. [CrossRef]
22. Xiaohua, L.; Kai, C. The research and application of IP multicast in enterprise network. In Proceedings of the International Conference on Internet Computing and Information Services, Hong Kong, China, 17–18 September 2011; pp. 191–194.
23. Andrews, G.R. Paradigms for process interaction in distributed programs. *ACM Comput. Surv.* **1991**, *23*, 49–90. [CrossRef]
24. Losada, N.; Bosilca, G.; Bouteiller, A.; González, P.; Martín, M. Local rollback for resilient mpi applications with application-level checkpointing and message logging. *Future Gener. Comput. Syst.* **2019**, *91*, 450–464. [CrossRef]