IEICE TRANS. INF. & SYST., VOL.E100–D, NO.1 JANUARY 2017

LETTER

Broadcast Network-Based Sender Based Message Logging for Overcoming Multiple Failures

Jinho AHN[†a)], ***Member***

SUMMARY All the existing sender-based message logging (SBML) protocols share a well-known limitation that they cannot tolerate concurrent failures. In this paper, we analyze the cause for this limitation in a unicast network environment, and present an enhanced SBML protocol to overcome this shortcoming while preserving the strengths of SBML. When the processes on different nodes execute a distributed application together in a broadcast network, this new protocol replicates the log information of each message to volatile storages of other processes within the same broadcast network. It may reduce the communication overhead for the log replication by taking advantage of the broadcast nature of the network. Simulation results show our protocol performs better than the traditional one modified to tolerate concurrent failures in terms of failure-free execution time regardless of distributed application communication pattern.

key words: *distributed system, rollback recovery, concurrent failures, sender-based message logging, broadcast network*

1. Introduction

All of the existing SBML protocols [3]–[6] have the same limitation that they can tolerate only a single failure at a time, called sequential failures. So, if more than one process crash concurrently, they may not make the entire system consistent, which is a critical shortcoming of the original SBML. This limitation comes from its operational procedure that a copy of the receive sequence number (RSN) of each message is kept only in the sender's volatile memory. In this paper, we designed an effective SBML protocol to have the following features. First, in order to tolerate concurrent failures while ensuring system consistency, the protocol enables each process to receive the log information of each message destined to another process on the same network and save the information into its volatile storage. This feature allows the execution of the entire system to progress without stopping and restarting it even with only one surviving process at a certain time. However, this redundancy may require a high extra communication overhead. In general, this functionality may be implemented in one of the following two communication modes, unicasting and broadcasting. Unicasting is the most typically-used mode where a sending node transmits a message only to a single receiver. This mode has an advantage that it can be used for non-broadcast network environments, but, if there are many nodes on a network to deliver the message at the same time, is significantly inefficient because a separate copy of the message should be sent exactly to each receiver. On the other hand, broadcasting is the mode of sending one single copy of the message to the network only once to be delivered by all destination nodes. Its usage may be dedicated to broadcast-capable network environments, but it can considerably improve communication performance by eliminating traffic redundancy and reducing network traffic control and node load compared to unicasting. However, all the previous SBML protocols have been oblivious to the underlying network, which does not provide any advantage for ensuring high scalability in a cluster system on a broadcasting network. Our new protocol can highly reduce the communication overhead resulting from the RSN replication by effectively utilizing the nature of broadcast networks without sacrificing the no rollback property.

Manuscript received June 20, 2016.
Manuscript revised September 22, 2016.
Manuscript publicized October 18, 2016.

[†]The author is with the Department of Computer Science, Kyonggi University, Suwon-si Gyeonggi-do, Korea.
a) E-mail: jhahn@kgu.ac.kr

DOI: 10.1587/transinf.2016EDL8127

2. The Proposed Novel SBML Protocol

Our SBML protocol is designed based on the following two observations. First, in order to satisfy no rollback property even in case of concurrent failures, replicating RSN of each message not only to volatile storages of its sender, but also of other processes except its receiver is essential. However, this redundancy generally requires a high extra communication overhead. Second, all the previous protocols are oblivious to the underlying network. This indifference may not provide any advantage for ensuring high scalability required in a cluster system composed of a large number of nodes that is based on a broadcasting network. In a broadcast communication environment, we found out the inherent limitation of SBML can be removed without any sacrifice of the no rollback property while minimizing the communication overhead resulting from the RSN replication by utilizing the nature of broadcast network. For this purpose, each process maintains the following data structures in our protocol.

• $SndLg_i$: a set saving $e(rcvr, ssn, rsn, data)$ of each message sent by i. Here, e is the log information of a message and the four fields are the identifier of the receiver, the send sequence number, the receive sequence number, and the data of the message, respectively.

• $RsnLg_i$: a set which maintains $e(sndr, ssn, rcvr, rsn)$ of each message received by i or another process on the network. Here, e is the log information of the message, where the four fields are the sender's id, the send sequence number, the receiver's id, and the receive sequence number of the message, respectively.

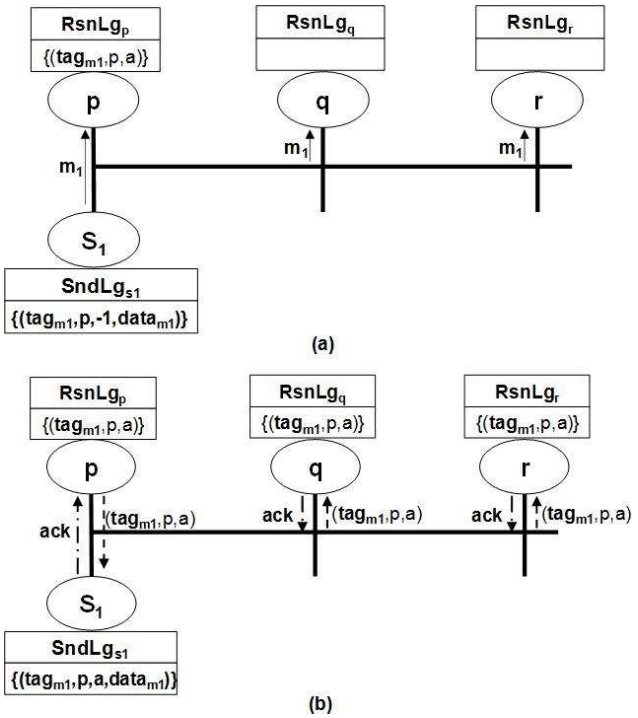


Fig. 1 Normal execution of our SBML protocol.

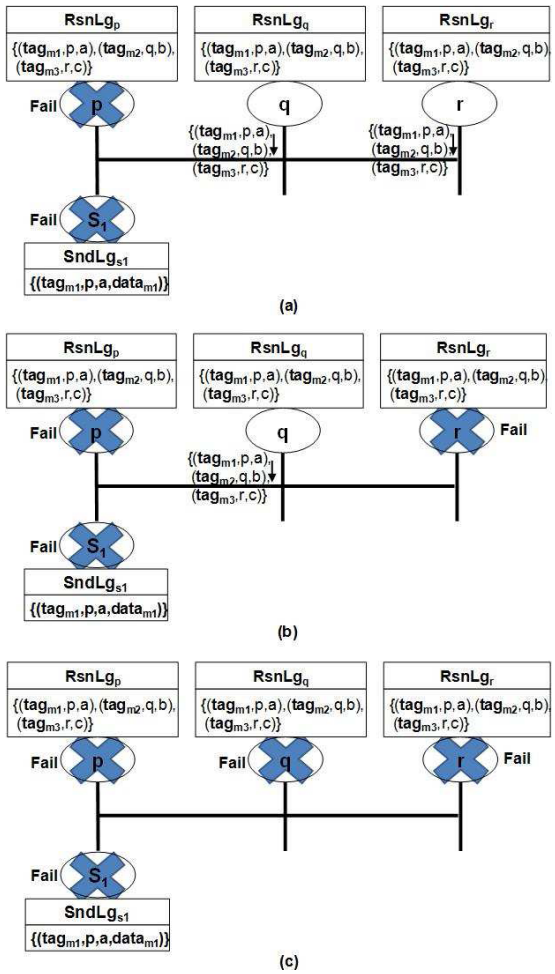


Fig. 2 Consistency satisfaction in concurrent failures.

Now, let us observe how the RSN replication process of our proposed protocol is performed in an effective way using Fig. 1. This figure shows the case where three messages, m_1, m_2 and m_3, are sent to p, q and r from senders S_1, S_2 and S_3 in this order through a broadcast network. First, when the first message m_1 is transmitted to p from S_1 as in Fig. 1 (a), the partial log information except for m_1's RSN is saved into $SndLg_{S1}$. At this time, as the message passes on the broadcast network, other nodes can receive it, too. So, if the latter, q or r, can afford to save the partial log information on its volatile log like S_1, the message may be buffered in it. This proactive approach can help p replay m_1 much faster during recovery by q's or r's giving p m_1's data directly. However, maintaining data of every message destined to another process passing on a broadcast network in volatile memory of each process may not be appropriate even for high-end computers. When p receives and assigns its current RSN (=a) to m_1, it transmits SID, SSN, RID and RSN of m_1 through the broadcast network and blocks sending all messages generated and sent from p after m_1 until p is informed that the replication process of m_1's RSN is completed by other processes on the same broadcast network. At this point, q and r as well as S_1 can catch the full log information of m_1 except for its data and keep them in its volatile memory like in Fig. 1 (b). Then, S_1, q and r should acknowledge their receipt of m_1's RSN, which ensures the k concurrent failure tolerance of SBML. Although this example is made for the case of tolerating n simultaneous failures, the number of concurrent failures may be parameterized according to how many degrees of failures at once the target system should tolerate. When p has received all acknowledgments from the others, it releases the blocking and sends all the messages generated after m_1 and before m_1's successor.

Second, let us identify how our protocol can address the limitation of the previous SBML in case of concurrent failures using Fig. 2 from m_1's perspective. Figure 2 (a) shows the case where both m_1's sender S_1 and its receiver p crash at the same time after our protocol has performed the same steps mentioned above with m_2 and m_3 from Fig. 1 (b). In this case, our protocol allows q and r to be able to offer log information of all the three messages to p during recovery, which can replay m_1 like in its pre-failure state and keep all log information for the other processes on its volatile log for preparing their concurrent failures. Even if r also crashes like in Fig. 2 (b), q can provide the same log information for both p and r during recovery. In the worst case, if every process on the same network would fail concurrently like in Fig. 2 (c), no inconsistency issue arises because even if m_1 without its original RSN value couldn't be replayed in its pre-failure position, this discordancy doesn't make any orphan messages sent to surviving processes depending on m_1.

Lemma 1. Our proposed protocol always prevents every

failed process p from making any other live process on the same network orphan.

Proof. We denote a set of all fully logged messages that p has received before its failure by $FULLY\text{-}LOGGED\text{-}MSGs_p$. The proof proceeds by induction on the number of all the messages in $FULLY\text{-}LOGGED\text{-}MSGs_p$, denoted by $NUMOF(FULLY\text{-}LOGGED\text{-}MSGs_p)$.

[Base cases]

We assume $NUMOF(FULLY\text{-}LOGGED\text{-}MSGs_p)$=1, *i.e.*, there is only one fully logged message m that p has received before its failure. There are two cases we should consider.

Case 1: m's sender s is a surviving process.

In this case, s can trivially give p all RSNs of m from $SndLg_s$ that p has assigned to m before. So, p can put them into $RsnLg_p$ and then, replay m in m's original order as in its pre-failure state. Therefore, no surviving process that has received any message sent from p after m's receipt becomes orphan.

Case 2: m's sender s has failed.

In this case, there are two sub-cases we should consider.

Case 2.1: there is no surviving process on the network.

In this case, even although p replays m in any order unlike in its pre-failure state, no inconsistency problem will occur in the system because there is no orphan state on which m's reception before its failure has any impact.

Case 2.2: there is at least one surviving process q.

In this case, as message m has been fully logged, m's receiver p before failure transmitted all surviving processes all rsns of m p had assigned to m before. So, p can get the RSNs from $RsnLg_q$ of q and put them into $RsnLg_p$, and then replay m in m's original order like in its pre-failure state by obtaining m's data from recovering s. Therefore, p's failure never makes any surviving process that has received any message sent from p after m's reception an orphan.

[Induction hypothesis]

We assume that the theorem is true for p in case that $NUMOF(FULLY\text{-}LOGGED\text{-}MSGs_p)$=$k$.

[Inductive step]

By induction hypothesis, p can get all the log information of k fully logged messages it has received before its failure. Therefore, if p can take the log information of (k+1)-th message fully logged before its failure in this recovery procedure, the theorem is true for p in case $NUMOF(FULLY\text{-}LOGGED\text{-}MSGs_p)$=$k$+1. The subsequent steps are similar to the base case stated above.

By the induction, among the lost state intervals of every failed process, all those that any normally operating process's state depends on can always be recovered by performing the proposed protocol. □

Theorem 1. Even if k ($1 < k \leq n$) processes on the same network crash concurrently, our proposed protocol allows consistent recovery to be completed.

Proof. We prove this theorem by contradiction. Assume that our protocol may not make consistent recovery possible in case of k simultaneous process failures. We denote the set of all failed processes by $SET\text{-}OF\text{-}FAILEDPROCs$ and the set of all surviving processes by $SET\text{-}OF\text{-}SURPROCs$. There are two process failure cases to consider as follows.

Case 1: there is no surviving process ($k = n$).

In this case, there occurs no orphan state interval, meaning every recovering process can replay each received message in a new order without considering the current state interval of any normally operational process.

Case 2: there are one or more surviving processes ($1 < k < n$).

This case means there is at least one orphan state interval their current states depend on directly or indirectly. This set of orphan state intervals can be classified into two kinds: state intervals directly created by the messages sent by any failed processes, $SET\text{-}OF\text{-}DIROSIs$, and directly created by the messages sent by any other surviving processes, $SET\text{-}OF\text{-}INDIROSIs$. In this case, there are two sub-cases to consider:

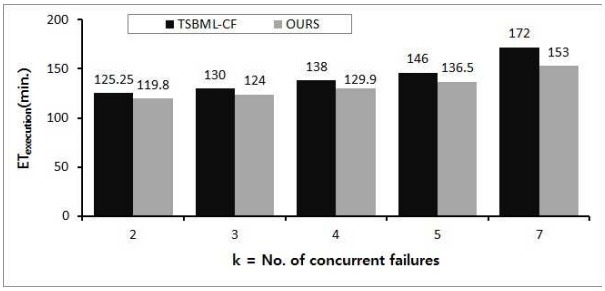
Case 2.1: Any state interval $si \in SET\text{-}OF\text{-}DIROSIs$ is created by the receive event of a message m. In this case, suppose si is created by $receive_p(m)$ at $p \in SET\text{-}OF\text{-}SURPROCs$ and depends on the receive events of all the messages that p has received until generating si including $receive_p(m)$. Even though all the senders of the received messages, denoted by $DirectMsgsSenders(si)$, would be a subset of $SET\text{-}OF\text{-}FAILEDPROCs$, by lemma 1, si never becomes an orphan state because the proposed protocol forces no crashed process $\in DirectMsgsSenders(si)$ to make p an orphan.

Case 2.2: Any state interval $si \in SET\text{-}OF\text{-}INDIROSIs$ is created by the receive event of a message m. In this case, if si depends transitively on any state interval $si' \in SET\text{-}OF\text{-}DIROSIs$, it may be an orphan state if si' could not be restored even after having completed the recovery procedure. But, this situation cannot occur according to case 2.1.

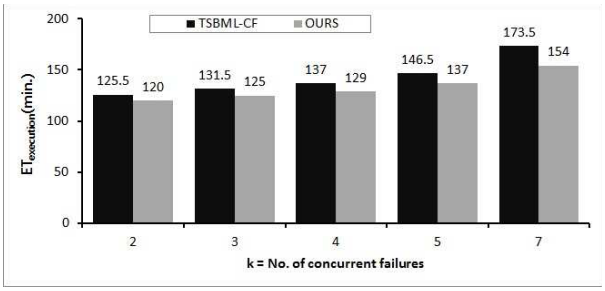
Therefore, consistent recovery is possible in all the cases. This contradicts the hypothesis. □

3. Evaluation and Concluding Remarks

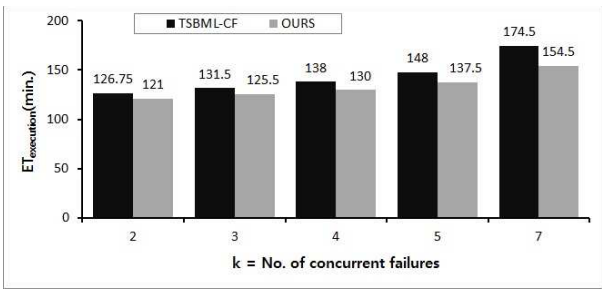
We have performed extensive simulations to evaluate performance of the two protocols, $TSBML\text{-}CF$ (Traditional SBML for Concurrent Failures) and our protocol, $OURS$, using a discrete-event simulation language named PARSEC [2]. In method $TSBML\text{-}CF$, when a receiver receives an application message m from a sender, it sends a separate message with m's RSN to k($1 \leq k < n$) other processes on the network including the sender. Here, k is the number of concurrent failures to attempt to tolerate. For this evaluation, there are two performance indices to be considered as follows. The first index ($ET_{execution}$) is measured for comparing the failure-free overhead, *i.e.*, the elapsed time until the same distributed execution has been completed. The second one ($Rate_{inc}$) is the increasing rate of the performance overhead of the two protocols for k simultaneous failures against the traditional SBML ($TSBML$) [5] for only tolerating a single failure at a time. A system of 10 nodes connected through a broadcast network is simulated.



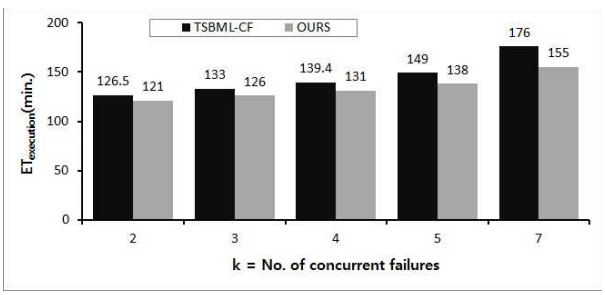
(a) Serial pattern.



(b) Circular pattern.



(c) Hierarchical pattern.



(d) Irregular pattern.

Fig. 3 $ET_{execution}$s for the four patterns.

Each node executes one process, and for simplicity, it is assumed that the processes are initiated and completed simultaneously. The degree of redundancy ranges from 3 to 8 for tolerating 2 through 7 process failures occurring at the same time, and the target of each application message sent from a process is always one process. Thus, IP multicast is used for multicasting a control message to return the RSN of each received message to a group of processes according to the number of failures required to tolerate in our protocol. 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 size of application messages ranges from 1KB to 1MB. Normal checkpointing is performed at each process periodically with the interval T_{nc}, following an exponential distribution with a mean value of 300 seconds. In addition, a message to a process is sent with an interval T_{ms}, following an exponential distribution with a mean value of 3 seconds. Distributed applications used for the simulation exhibit the four communication patterns respectively [1]. The results from the simulations are averaged over multiple trials. Figure 3 shows $ET_{execution}$ for the two protocols, *TSBML-CF* and *OURS*, for four different communication patterns respectively with varying number of concurrent failures, k, required to tolerate in case the total number of processes is 10. As k grows, $ET_{execution}$ values in both protocols increase as well in all four patterns because the RSN replication overhead becomes larger. However, the figures indicate that the $ET_{execution}$ value in *OURS* is much lower than in *TSBML-CF* regardless of the application communication patterns. The reduction of $ET_{execution}$ value in *OURS* over *TSBML-CF* ranges from 3.2% to 11.9%. The reason is the *OURS* protocol uses only one control message for returning the RSN of each received message to a group of processes whereas the *TSBML-CF* protocol uses multiple messages directly proportional to the number of failures required to tolerate.

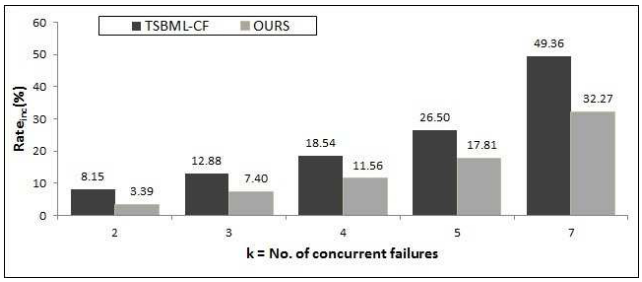


Fig. 4 Performance overhead.

Figure 4 shows the average values of $Rate_{inc}$ of the two protocols, *TSBML-CF* and *OURS*, for the four communication patterns when k ranges from 2 to 7. In this case, the average $ET_{execution}$ value of *TSBML* is 116.5 minutes. As k becomes bigger, their $Rate_{inc}$ values are also increasing. This phenomenon arises from the reason that the increase of the number of concurrent failures to tolerate leads to the higher interaction overhead resulting from the RSN replication procedure. But, this simulation results illustrate the differences in $Rate_{inc}$ values in both protocols grow as the value of k increases. Especially, when k is less than 5, $Rate_{inc}$ of *OURS* isn't greater than 12% unlike *TSBML-CF*. From

this outcome, it may be claimed that our protocol incur the reasonable performance overhead against $TSBML$ in order to tolerate simultaneous failures in this range.

From these observations, we conclude that $OURS$ significantly reduces the communication cost resulting from the RSN replication compared with $TSBML\text{-}CF$ by utilizing the nature of broadcast networks in any communication pattern.

Acknowledgments

This work was supported by Kyonggi University Research Grant 2014 (Project Number: 2014-012).

References

[1] G.R. Andrews, "Paradigms for process interaction in distributed programs," ACM Computing Surveys, vol.23, no.1, pp.49–90, 1991.

[2] 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, no.10, pp.77–85, 1998.

[3] B. Gupta, R. Nikolaev, and R. Chirra, "A recovery scheme for cluster federations using sender-based message logging," J. Comp. and Info. Tech., vol.19, no.2, pp.127–139, 2011.

[4] P. Jaggi and A. Singh, "Log based recovery with low overhead for large mobile computing systems," J. Info. Sci. and Eng., vol.29, no.5, pp.969–984, 2013.

[5] D. Johnson and W. Zwaenpoel, "Sender-based message logging," In Proc. of the 7th International Symposium on Fault-Tolerant Computing, pp.14–19, 1987.

[6] E. Meneses, C. Mendes, and L. Kalé, "Team-based message logging: preliminary results," In Proc. of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, pp.697–702, 2010.