Enhanced Sender-Based Message Logging for Reducing Forced Checkpointing Overhead in Distributed Systems

Jinho AHN\(^{(a)}\), Member

SUMMARY The previous communication-induced checkpointing may considerably induce worthless forced checkpoints because each process receiving messages cannot obtain sufficient information related to non-causal Z-paths. This paper presents an enhanced sender-based message logging protocol applicable to any communication-induced checkpointing to lead to a high decrease of the forced checkpointing overhead of communication-induced checkpointing in an effective way while permitting no useless checkpoint. The protocol allows each process sending a message to know the exact timestamp of its receiver in logging procedures without any extra message. Simulation verifies their great efficiency of overhead alleviation regardless of communication patterns.

key words: distributed systems, fault-tolerance, rollback-recovery, checkpointing, message logging

1. Introduction

To harmonize the pros and cons of uncoordinated and coordinated checkpointing, communication-induced checkpointing (CIC) was developed to provide best-effort autonomy with respect to taking local checkpoints while preventing them from being useless for recovery with forced checkpoints\(^{[8], [9]}\). The prior research works on CIC with this feature have pursued a ultimate goal to minimize the number of forced checkpoints\(^{[8], [9], [15]}\). HMNR protocol\(^{[9]}\) effectively utilizes some control information vectors contained in each sent message to attempt to decrease the number of worthless forced checkpoints to a maximum. Afterwards, an improved HMNR protocol, LazyHMNR\(^{[14]}\), uses a lazy indexing strategy to alleviate high frequency of forced checkpointing which may occur in some particular cases\(^{[8]}\). However, even a family of HMNR protocols\(^{[8], [9], [14]}\) has each process forced to take extra checkpoints more than twice the number of basic checkpoints\(^{[2]}\). The previous CIC may significantly induce useless forced checkpoints because each process receiving messages cannot obtain sufficient information related to non-causal Z-paths. To the best of our knowledge, there is no work to develop protocols to lower the number of forced checkpoints in an unconventional way more than HMNR and LazyHMNR while permitting no useless checkpoint.

Sender-based message logging(SBML)\(^{[1], [6], [11]}\) has been widely used as a lightweight rollback-recovery technique that can be combined with checkpointing-based recovery including CIC for highly reducing the rollback distance of each process during recovery. This popularity comes from considerably alleviating the normal operation overhead of synchronous logging on stable storage by volatile logging at sender’s memory\(^{[1], [6], [7], [11]}\). Especially, one of the most recent SBML protocols, ScalableSBML\(^{[1]}\), is designed to significantly decrease the number of control messages needed for volatile logging at message senders. It has the feature that each receiver delays the update of receive sequence numbers of a sequence of messages consecutively received to their senders until there comes out the first message it is willing to send\(^{[1]}\). However, all the existing SBML protocols including ScalableSBML embody no feature to lower the number of useless forced checkpoints that may be induced by the conventional CIC including HMNR and LazyHMNR. This paper proposes an enhanced sender-based message logging(ESBML) protocol to considerably reduce frequency of needless forced checkpointing in an efficient way. For this purpose, the protocol lets a process sending a message know the current local timestamp of its receiver in logging procedures of the original SBML with no extra message. The beneficial feature makes each process capable of detecting the situations where forced checkpointing isn’t required much earlier than the previous ones. It is applicable not only to a family of HMNR protocols, but also the other CIC ones to achieve this goal.

2. Preliminaries

2.1 System Model

A distributed computation \(\phi\) consists of a set \(P\) of \(n(n > 0)\) sequential processes executed on nodes in the system, and there is a stable storage that every process can always access that persists beyond processor failures\(^{[1], [6], [7]}\). Processes have no global memory and global clock. The system is asynchronous. Exchanging messages may temporarily be lost but, eventually delivered in first-in first-out order. We assume that the communication network is immune to partitioning and nodes fail according to the fail stop model\(^{[13]}\). A process \(p\) starting at its initial state, \(s^p_0\), generates a sequence of events, \(e^p_1, e^p_2, \ldots, e^p_k\). These events are classified into internal, sending, delivery. Process \(p\) may produce an internal event to execute a particular computation without any communication. A sending event of a message \(m\), de-
noted 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 \( e^p_1, e^p_2, \ldots, e^p_n \) to \( s_0^p \) in order, process $p$ has a unique local state $s^p_{i+k}(k \geq 0)$. Events of processes occurring in a failure-free execution are ordered using Lamport’s happened before relation [10].

Let the $i$-th local checkpoint of process $p$ denoted by $C^p_i$, which corresponds to a local state $s^p_i(k \geq i)$. It is supposed that each process $p$ takes its initial local checkpoint $C^p_0$ for saving $s^p_0$ when starting its execution. A set of local checkpoints, which consists of only one from every process in the system, is called a global checkpoint [7]. A message $m$ which $p$ transmits to $q$ is called orphan message [1, 6, 7] with respect to a pair of local checkpoints($C^p_k$, $C^q_j$) if and only if delivery($m$) in $C^p_j$ but, sending($m$) $C^q_j$. A global checkpoint is consistent if and only if any pair of local checkpoints in it is mutually consistent [5, 7]. To detect whether an ordered pair of checkpoints is mutually consistent, the notion of Z-path [12] is used to identify causal paths as well as non-causal (NC) paths which exist between the two checkpoints. A Z-path forming from a local checkpoint $C^p_j$ to the same checkpoint $C^p_j$ is Z-cycle [12].

**Theorem 1.** For any pair of checkpoints $C^p_i$ and $C^q_j$, if there is a Z-path starting from $C^p_i$ and ending with $C^q_j$, and the timestamp of the first, $C^p_i$, is less than that of the other, $C^q_j$, there exists no Z-cycle [9, 12].

### 2.2 Limitations of HMNR

HMNR [9] is designed to prevent any checkpoint from being useless while lowering the number of forced checkpoints to a maximum. To hold the feature in HMNR, each process $p$ should always have the following state information:

- $lc_p$: an integer variable which is the current value of $p$’s local clock and initialized to 0. If $p$ takes a local checkpoint, $lc_p$ increments by one and its value is assigned to the checkpoint as timestamp. As a message $m$ is sent from $p$, $lc_p$ is piggybacked on $m$. When $p$ receives message $m$ and $lc_p$ is less than the most recent value of the local clock of $m$’s sender (denoted by $m.lc$), $lc_p$ is updated with the latter. If $p$ sends the first message to $q$ since its latest checkpoint. Whenever $p$ takes a local checkpoint, all $send.to_q[p]$s are reset to $F$. If $p$ receives a message $m$, it checks whether there is at least one process $q$ for which the element $send.to_q[p]$ is $T$. If so, it means it is possible to exist at least one non-causal Z-path starting with $m$ after its current checkpoint and connecting with $q$.
- $ckpt_p$: a vector where $ckpt_p[j]$ is the number of checkpoints which $q$ has taken from the beginning perceived by $p$, and initialized to 0. If $p$ takes a local checkpoint, $ckpt_p[p]$ increments by one. When sending a message $m$, the message carries $ckpt_p$. If $p$ receives $m$, it checks whether the value of its current checkpoint identifier($ckpt_p[p]$) equals that of the most recent checkpoint identifier for $m(ckpt[p])$ piggybacked on $m$. If so, it means there exists at least one causal Z-path created after $m$’s current checkpoint and ending with $m$ before its next checkpoint. Before $m$ is actually delivered to $p$, each $ckpt_p[q](p\neq q)$ is updated to $m$.ckpt[$q$] if $m.ckpt[q]$ is greater than $ckpt_p[q]$.
- $taken_p$: a vector where $taken_p[q]$ is the boolean value for $q$ for recognizing a causal Z-path from the most recent checkpoint of $q$ known by $p$ to $p$’s next checkpoint. Whenever $p$ takes a local checkpoint, $taken_p[q](p\neq q)$ is set to $T$. But, $taken_p[p]$ is initialized to $F$ and remains unchanged. As a message $m$ is sent from $p$, $m$ includes $taken_p$. If $p$ receives $m$, it checks whether $m.taken[p]$ is $T$ when $ckpt_p[p]$ equals $m.ckpt[p]$. If so, the causal Z-path includes at least one Z-cycle and $p$ should take a forced checkpoint before delivering $m$ for ensuring the always-no-useless checkpoint condition. Afterwards, $taken_p[q](p\neq q)$ is updated with $m.taken[p]$ in proper ways depending on the result of the comparison between $ckpt_p[q]$ and $m.ckpt[p]$.
- $greater_p$: a vector where $greater_p[q]$ is the boolean value for $q$ to indicate whether $p$’s current timestamp $lc_p$ is greater than the most recent timestamp of $q$ perceived by $p(=T)$ or not($=F$). $greater_p[p]$ is initialized to $F$. Whenever $p$ takes a local checkpoint, all $greater_p[q]s(p\neq q)$ become $T$. When $p$ sends a message $m$, $greater_p$ is piggybacked on $m$. When $p$ receives a message $m$ from $q$, if $m.lc$ is greater than $lc_p$, $greater_p[q](p\neq q)$ is set to $m.greater[q]$ or if $m.lc$ is equal to $lc_p$, $greater_p[q]$ is set to $m.greater[q] \land m.greater[q]$. In both cases, $greater_p[p]$ is also set to $F$.

For this purpose, it uses Lamport’s logical clock-based checkpoint timestamping method [5]. Thanks to this feature, the protocol incurs no Z-cycle formation on causal Z-paths. However, the timestamping method cannot prevent NC Z-paths from being transformed into Z-cycles. HMNR attempts to avoid this situation by having each process $p$ receiving a message $m$ perform forced checkpointing if the following condition $C_{HMNR}$ is satisfied.

- $C_{HMNR} \equiv C_1 \lor C_2$
- $C_1 \equiv 3(1 \leq j \leq n): sent.to_q[j] \land m.greater[j] \land (m.lc > lc_p)$
- $C_2 \equiv (ckpt_p[p] = m.ckpt[p]) \land m.taken[p]$

The NC Z-path pattern satisfying $C_2$ is the unambiguous situation most of the existing CIC protocols including HMNR and LazyHMNR should take forced checkpoints to prevent basic checkpoints from being useless. However, the pattern satisfying $C_1$ may make the protocols considerably induce worthless forced checkpoints because each process receiving messages cannot obtain sufficient information related to NC Z-paths, resulting in high failure-free overhead. To reduce the number of forced checkpoints in this pattern, we propose ESBML to be applicable not only to a family of
HMNR, but also the other CIC.

2.3 Logging and Recovery of SBML

SBML [1], [6], [11] can considerably reduce the overhead for heavyweight synchronous logging on a stable storage by using volatile logging at sender’s memory. When a process p sends a message m to q, it saves a log element for m, $Sendlg_p(m)$, consisting of the receiver’s identifier(rcvr), the send sequence number(ssn), the receive sequence number(rsn) and data of m, into its volatile memory. At this point, the value of rsn of m is initialized to a null value(⊥) because it isn’t still determined. On receiving m, q increments and assigns its rsn $r_{sn_q}$ to m and sends an acknowledgment with $r_{sn_q}$ to its sender p. Then, p completes the log element for m by updating its rsn with the value of $r_{sn_q}$ piggybacked on the acknowledgment and sends a confirmation to q to inform of the reception of m’s rsn, which ensures the always-no-rollback property of SBML [1], [6], [11]. If q fails, q can get the log element for m from p and replay m in the same rsn order in its pre-failure execution [1], [6].

Let us see an example in Fig. 1 for understanding message logging and recovery procedures of SBML in details. In this figure, after having taken its latest checkpoint $C_k$, $p$ sends a message $m_1$ and $m_2$, from $p$ and $r$, sends a message $m_3$ to $r$, and then crashes. Without SBML, during recovery, $q$ may not replay the receipts of $m_1$ and $m_2$ in the same order like in the pre-failure state because $q$ cannot get the rsn of $m_1$ and $m_2$ anywhere and so, may restore to be in the state that it cannot regenerate $m_1$. In order to address the inconsistency issue, SBML performs as follows. When sending $m_1$ to $q$, $p$ saves $Sendlg_p(m_1)$ into its volatile storage. Likewise, when $r$ transmits $m_2$, $Sendlg_r(m_2)$ is recorded into its volatile storage. As $q$ receives $m_1$ and then $m_2$ in order, it increments and assigns their rsn, $r_{sn+1}$ and $(a+2)$, to them, and then sends each an acknowledgment, $ack_1$ with $(a+1)$ and $ack_2$ with $(a+2)$, to their corresponding senders respectively. As soon as $p$ and $r$ obtain rsn of the two messages from $q$, they update $Sendlg_p(m_1)$ and $Sendlg_r(m_2)$ with the rsn and then, confirms their rsn receipt to $q$ with cf$m_1$ and cf$m_2$ respectively. Suppose $q$ fails after sending $m_3$. In this case, after broadcasting each a recovery message to the other processes, $q$ can collect the rsn of both $m_1$ and $m_2$ from $p$’s and $r$’s volatile logs, $Sendlg_p$ and $Sendlg_r$, and so replay their receipts in the same order and reach the state where it can regenerate $m_3$ like in the figure.

3. Enhanced Sender-Based Message Logging

In this section, the proposed protocol is applied into HMNR to clearly explain its detailed procedures and effectiveness. LazyHMNR can also use it with a few of adaptations. HMNR must take forced checkpoints in the two NC Z-path patterns like in Fig. 2 if $C_1$ is satisfied because $C_2$ is always unsatisfied in both patterns. In the first pattern like in Fig. 2(a), when $q$ receives a message $m_1$, it can detect message $m_2$ is the first message sent to $r$ before $m_1$ after its latest checkpoint $C_k$. However, $q$ cannot know that the timestamp of $r$’s next checkpoint, $C_r$, is greater than $q$’s current timestamp $lc_q(=2)$ when sending $m_2$. Due to the lack of information, the value of $lc_q$ remains unchanged until $q$ receives $m_1$. When $p$ sends $m_1$ to $q$, greater$p$_$r(=m_1.greater[r]) and $lc_p(=3)$ > $lc_q(=2)$ are all T. Therefore, $q$ wrongly decides $C_r$ $lc$ $C_r$ $lc(C_1=\text{T})$, and so takes a forced checkpoint and deliver $m_1$ to its application.

The second pattern like in Fig. 2(b), is the case that there is a message $m_1$ which causally connects $m_2$ and $m_1$ before $r$ takes its next checkpoint $C_{k+1}$. In this case, although $p$ can get $lc_p(m_3,lc=2)$ through $m_3$, $p$ decides $C_r$ $lc$ $C_r$ $lc(C_1=\text{T})$ in HMNR because $lc_p(=3)$ > $m_3$. This situation has $p$ perform no update on $lc_p$ and greater$p$_$r(=\text{T})$. Therefore, when $q$ receives $m_1$, it takes an additional checkpoint before delivering $m_1$ as $sent_fo_q[r]$, $m_1.greater[r]$ and $m_1.lc > lc_q$ are all T($C_1=\text{T}$).

In order to alleviate the limitations of a family of HMNR protocols, ESBML has each process keep the following information.

- $r_{sn_q}$: the receive sequence number of the last message delivered to $p$.
- $ssn_p$: the send sequence number of the latest message sent by $p$.
- $Sendlg_p$: a set that records an element $e(rcvr, ssn, rsn, data)$ of each message sent by $p$.
- $S snVec_p$: It is a vector where $S snVec_p[q]$ is the ssn of the last message that was delivered to $p$ from a process $q$. It is used for message duplication detection.
- $RsnSet_p$: a set which maintains $e(sndr, ssn, rsn, lc)$ of each message received by $p$. Here, $e.lc$ is the value of the local clock when the corresponding message is delivered. It
is used for providing a recovering process sndr with both rsn and lc of the message.

It is developed by integrating and modifying procedures of both SBML and HMNR as follows. When \( p \) sends \( m \) to \( q \) (by executing Module Send() in Fig. 3), ESBML allows \( q \) to piggyback \( lc \) as well as \( m \)'s rsn on \( m \)'s acknowledgment \( \text{ack} \) (by executing Module Recv() in Fig. 3). When \( p \) receives \( \text{ack} \), it updates \( lc_p \) with \( \text{ack}.lc \) if \( \text{ack}.lc > lc_p \) (by executing Module Recv-Ack() in Fig. 3). In this case, \( greater_p[q] \) is set to \( F \) and \( \forall w(1 \leq w \leq n, w \neq q, w \neq p): greater_p[w] = T \). If \( \text{ack}.lc = lc_p \), \( greater_p[q] \) is just set to \( F \). Next, \( p \) sends a confirmation to \( q \) to ensure the reception of \( m \)'s rsn and \( lc \). In this case, \( greater_q[p] \) is set to \( F \) (by executing Module Recv-Confirm() in Fig. 3). For example, in Fig. 2(a), when \( r \) receives \( m_2 \) from \( q \), it doesn’t change its variables and sends to \( q \) \( m_2 \)'s acknowledgment \( \text{ack2} \) with \( m_2 \)'s rsn and \( lc \). As \( \text{ack2}.lc > lc_q \), \( q \) updates \( lc_q, greater_r[r] \) and \( greater_q[p] \) to \( ack2.lc \) and \( F \) and \( T \) respectively. When \( r \) receives \( \text{ack2} \)'s confirmation, it sets \( greater_r[r] \) to \( F \). When \( q \) receives \( m_1 \) from \( p \), \( C_1 \) becomes \( F \) as \( m_1.greater[r]=T \), but \( m_1.lc=lc_q=3 \). Therefore, in ESBML, \( q \) takes no forced checkpoint before delivering \( m_1 \) as it can know \( C_k^{k+1} < C_k^{k+1}.lc \). Then, only \( greater_q[p](\neg m_1.greater[p] \lor greater_q[p]) \) is set to \( F \). Also,

\[ \text{greater}_p[q] = \text{greater}_p[q] \top (m_2.greater[p] \lor \text{greater}_q[p]) \]

When \( p \) initializes its variables and takes its first checkpoint before starting its computation.

Module Initialize()

\[
\text{take}_p[p] \leftarrow F; \quad \text{greater}_p[p] \leftarrow F; \quad lcp \leftarrow 0; \quad rsn_p \leftarrow 0; \quad \forall(1 \leq j \leq n): \text{ckpt}_p[j] \leftarrow 0;
\]

call Module Take-Checkpoint();

When \( p \) sends a message \( m \) to \( q \)

Module Send(data, q)

\[
\text{if}(\text{sent}_\text{top}[q] = \text{F}) \text{ then } \text{sent}_\text{top}[q] \leftarrow \text{T}; \\
\text{ssn}_p \leftarrow \text{ssn}_p + 1; \\
\text{send}(\text{msg}_p, data, lcp, \text{greater}_p, \text{ckpt}_p, \text{taken}_p) \rightarrow q; \\
\text{Send}(q \leftarrow \text{Send}(q \cup \{(q, \text{ssn}_p, 1, \text{data})\}); \\
\]

When \( p \) receives a message \( m \) from \( q \)

Module Recv(msg, data, lc, greater, ckpt, taken) from \( q \)

Check whether \( m \) is a duplicate message for recovery.

\[
\text{if}(\text{SSN Vec}_p < \text{m.ssn}) \text{ then } \\
\text{rsn}_q \leftarrow \text{rsn}_q + 1; \\
\text{SSN Vec}_p[q] \leftarrow \text{m.ssn}; \\
\]

Send an acknowledgment with \( m \)'s rsn and \( lc \).

\[
\text{if}(lcp > m.lc) \text{ then } \text{send}(m.ssn, \text{rsn}_q, m.lc) \rightarrow q; \\
\text{else send}(m.ssn, \text{rsn}_q, lcp) \rightarrow q; \\
\text{save all the messages produced to send after } m \text{ into } \\
\text{a buffer tagged with } rsn;p; \\
\]

Check if a forced checkpoint should be taken before \( m \).

\[
\text{if}(\exists (j \leq l \leq n): \text{sent}_\text{top}[j] \land \text{greater}_j[lc, m.lc]) \lor \\
\text{((\text{ckpt}_p[p] \text{= } m.ckpt) \land \text{m.taken}[p]))} \text{ then } \\
call Module Take-Checkpoint(); \\
\]

Update its local clock and Z-cycle detection variables.

\[
\text{if}(m.lc > lcp) \text{ then } \\
\text{greater}_p[p] \leftarrow F; \quad lcp \leftarrow m.lc; \\
\forall(1 \leq j \leq n, j \neq p): \quad \text{greater}_j[p] \leftarrow m.greater[j]; \\
\text{else if}(lcp = m.lc) \text{ then } \\
\forall(1 \leq j \leq n): \quad \text{greater}_j[p] \leftarrow m.greater[j]; \\
\forall(1 \leq j \leq n, j \neq p); \\
\text{if}(\text{m.ckpt} \geq \text{ckpt}_p[j]) \text{ then } \\
\text{ckpt}_p[j] \leftarrow m.ckpt[j]; \quad \text{taken}_p[j] \leftarrow m.taken[j]; \\
\text{else if}(\text{m.ckpt} < \text{ckpt}_p[j]) \text{ then } \\
\text{taken}_p[j] \leftarrow m.taken[j]; \\
\text{RSN Set}_p \leftarrow \text{RSN Set}_p \cup \{(q, m.ssn, \text{rsn}_p, lcp)\}; \\
\text{deliver } m.\text{data} \text{ to its corresponding application}; \\
\text{else } \\
\text{Send an acknowledgment with } m \text{'}s rsn and } lc \text{ to } q \text{ for recovery. } \\
\text{find } 3e \in \text{RSN Set}_p, \text{st}(e.\text{sndr} = q) \land (e.\text{ssn} = m.ssn)); \\
\text{send}(m.ssn, e.rsn, e.lc) \rightarrow q; \\
\]

When \( p \) receives an acknowledgment from \( q \)

Module Recv-Ack(ack, rsn, lc) from \( q \)

\[
\text{find } 3e \in \text{Send}(q \rightarrow \text{st}(e.\text{recv} = q) \land (e.\text{ssn} = \text{ack}.ssn)); \\
\text{e.rsn} \leftarrow \text{ack}.rsn; \\
\]

Update its local clock and Z-cycle detection variables.

\[
\text{if}(lcp > lc) \text{ then } \\
\text{greater}_p[q] \leftarrow F; \quad lcp \leftarrow lc; \\
\forall(1 \leq j \leq n, j \neq p, j \neq q): \quad \text{greater}_j[p] \leftarrow T; \\
\text{else if}(lcp = lc) \text{ then } \text{greater}_p[q] \leftarrow F; \\
\text{send}(\text{cfm}(\text{ack}.rsn)) \rightarrow q; \\
\]

When \( p \) receives a confirmation from \( q \)

Module Recv-Confirm(cf m(srn)) from \( q \)

allow all the send message operations delayed after receiving \( m \) before its successor to begin executing;

\[
\text{greater}_p[q] \leftarrow F; \\
\]

When \( p \) takes its local checkpoint on the stable storage

Module Take-Checkpoint()

\[
\text{lcp} \leftarrow lcp + 1; \quad \text{ckpt}_p[p] \leftarrow \text{ckpt}_p[p] + 1; \\
\forall(1 \leq j \leq n): \text{sent}_\text{top}[j] \leftarrow F; \\
\forall(1 \leq j \leq n, j \neq p): \quad \text{greater}_j[p] \leftarrow T; \quad \text{taken}_p[j] \leftarrow T; \\
take its local checkpoint with (lcp, \text{ssn}_p, \text{rsn}_p, \text{SSN Vec}_p, \\
\text{Send}(q \rightarrow \text{Send}(q \cup \{(q, \text{ssn}_p, 1, \text{data})\}); \\
\text{RSN Set}_p \leftarrow \emptyset; \\
\]

Fig. 3 Procedures for process \( p \) in ESBML.

(a) The first NC Z-path pattern

(b) The second NC Z-path pattern

Fig. 2 How to alleviate forced checkpointing in two patterns
when \( p \) receives \( \text{ack}_k \) from \( q \)(although not shown in this figure), only \( \text{greater}_p[q] \) is updated to \( F \) because \( \text{ack}_k.l_c = l_c_p \).

Also, in the second pattern like in Fig. 2(b), when \( p \) receives \( m_1 \) from \( r \), ESBML has \( p \) send \( \text{ack}_k \) with both \( m_1 \)’s \( r_{sn} \) and \( l_{cp} \) to \( r \). In this case, as \( \text{ack}_k.l_c(=3) > l_c_r(=2) \), \( r \) lets \( l_{cr} = \text{ack}_k.l_c \) and \( \text{greater}_r[p] \) ← \( F \) and changes \( \text{greater}_r[q] \) to \( T \). When \( p \) receives a confirmation of \( \text{ack}_k \) from \( r \), \( \text{greater}_r[r] \) is set to \( F \). When receiving \( m_1 \), \( q \) decides \( C_1 \) is \( F \) because \( \text{sent.to}_q[r]=T \), \( m_1.l_c(=3) > l_{cq}(=2) \), but \( m_1.greater[r] = F \). Therefore, as \( q \) can know \( Ck_q^1.l_c < Ck_r^{k+1}.l_c \), it takes no forced checkpoint before delivering \( m_1 \), and sets both \( \text{greater}_q[p] \) and \( \text{greater}_q[r] \) to \( F \) and updates \( l_{cq} \) with \( m_1.l_c \). The detailed formal description of our protocol is shown in Fig. 3.

**Theorem 2.** ESBML with HMNR ensures no checkpoint is useless.

(the proof is omitted due to space limitation)

### 4. Simulation

We have performed extensive simulations to evaluate performance of the two protocols, LazyHMNR [14] and E-HMNR, using a discrete-event simulation language [4]. LazyHMNR is a most recently developed version of HMNR for decreasing high frequency of forced checkpointing as follows [8]. If the checkpointing frequency of one particular process increases fast, its checkpoint timestamp becomes much larger accordingly. When other processes receive messages from the first, they may unnecessarily take forced checkpoints. To alleviate this shortcoming, the strategy makes each process suppress increasing the timestamp of its next checkpoint until the process receives any message having a timestamp higher than or equal to its current one. E-HMNR is our enhanced version of LazyHMNR with ESBML.

We use one major performance index for this evaluation to consider as follows. The index, \( \text{ratio}_NOFC \), is the ratio of the total number of forced checkpoints of LazyHMNR to that of E-HMNR. A system with \( N \) nodes connected through a broadcast network is simulated. Each node has one process executing on it and, for simplicity, the processes are assumed to be initiated and completed together. The target of each application message sent from a process is always one process. The message transmission capacity of a link in the network is 100 Mbps and its propagation delay is 1 ms. Every process has a 128 MB buffer space for storing its message log. The size of application messages ranges from 1 KB to 100 KB. Normal checkpointing is initiated at each process with an interval following an exponential distribution with a mean of \( T_{mc} = 300 \) seconds. In addition, each message 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 four communication patterns [3].

Figure 4 shows \( \text{ratio}_NOFC \) for the two protocols with varying the number of processes, \( NOP \), ranging from 6 to 12 for four different communication patterns. As \( NOP \) grows, \( \text{ratio}_NOFC \) goes up from 2.95 to 5.2 in all four patterns because the possibility of forming the two kinds of NC Z-path patterns in Fig. 2 where \( C_1 \) is satisfied in LazyHMNR becomes higher. These results mean ESBML can make each process know the timestamps of the first and last checkpoints on the two sorts of NC Z-paths, like \( Ck_q^1 \) and \( Ck_r^{k+1} \) in Fig. 2, much earlier and more exactly than LazyHMNR. Also, it has the effect of high reduction of forced checkpoints regardless of the communication patterns, but the degree of their effectiveness may fluctuate in the irregular pattern because its irregularity may make frequency of occurrences of the two patterns change every run.

### 5. Conclusion

In this paper, ESBML with HMNR is designed to lead to much higher decrease of forced checkpointing overhead of communication-induced checkpointing in an effective way than the previous ones while permitting no useless checkpoint. It allows each process sending a message to know the exact timestamp of its receiver in its logging procedures without any extra message. Simulation results confirm their great efficiency of overhead alleviation regardless of communication patterns. Also, ESBML is applicable not only to a family of HMNR, but also the other CIC protocols to highly reduce the number of forced checkpoints.

### Acknowledgments

This work was supported by Kyonggi University Research Grant 2018 (Project Number: 2018-034).

### References

[1] J. Ahn, “Scalable sender-based message logging protocol with little communication overhead for distributed systems,” Parallel Process.
[2] L. Alvisi, E. Elnozahy, S. Rao, S.A. Hussain, and A. de Mel, “An analysis of communication induced checkpointing,” Proc. 29th Int. Symp. Fault-Tolerant Computing, pp.242–249, June 1999.
[3] G.R. Andrews, “Paradigms for process interaction in distributed programs,” ACM Comput. Surv., vol.23, no.1, pp.49–90, March 1991.
[4] R. Bagrodia, R. Meyer, M. Takai, Y. Chen, X. Zeng, J. Martin, and H.Y. Song, “Parsec: a parallel simulation environments for complex systems,” Comput J., vol.31, no.10, pp.77–85, Oct. 1998.
[5] K.M. Chandy and L. Lamport, “Distributed snapshots: determining global states of distributed systems,” ACM Trans. Comput. Syst., vol.3, no.1, pp.63–75, Feb. 1985.
[6] D. Johnson and W. Zwaenpoel, “Sender-based message logging,” Proc. 7th International Symp. on Fault-Tolerant Comput., pp.14–19, 1987.
[7] E. Elnozahy, L. Alvisi, Y. Wang, and D. Johnson, “A survey of rollback-recovery protocols in message-passing systems,” ACM Comput Surv., vol.34, no.3, pp.375–408, Sept. 2002.
[8] I.C. Garcia, G.M.D. Vieira, and L.E. Buzato, “A rollback in the history of communication-induced checkpointing,” submitted (arXiv:1702.06167 [cs.DC]), June 2019.
[9] J.-M. Hélary, A. Mostefaoui, R.H.B. Netzer, and M. Raynal, “Communication-based prevention of useless checkpoints in distributed computations,” Distrib. Comput., vol.13, no.1, pp.29–43, 2000.
[10] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” Commun. ACM, vol.21, no.7, pp.558–565, July 1978.
[11] H. Meyer, R. Muresano, M. Castro-Leon, D. Rexachs, and E. Luque, “Hybrid message pessimistic logging: improving current pessimistic message logging protocols,” J. Parallel. Distrib. Comput., vol.104, no.C, pp.206–222, June 2017.
[12] R.H.B. Netzer and J. Xu, “Necessary and sufficient conditions for consistent global snapshots,” IEEE Trans. Parallel Distrib. Syst., vol.6, no.2, pp.165–169, Feb. 1995.
[13] R.D. Schlichting and F.B. Schneider, “Fail-stop processors: an approach to designing fault-tolerant distributed computing systems,” ACM Trans. Comput. Syst., vol.1, no.3, pp.222–238, Aug. 1985.
[14] J. Tsai, “Applying the fully-informed checkpointing protocol to the lazy indexing strategy,” J. Inf. Sci. Eng., vol.23, no.5, pp.1611–1621, 2007.
[15] M. Vargas-Santiago, L. Morales-Rosales, R. Monroy, S. Pomares-Hernandez, and K. Drira, “Autonomic web services based on different adaptive quasi-asynchronous checkpointing techniques,” Appl. Sci., vol.10, no.7, pp.2495–2515, 2020.