electronics

Article

Communication-Induced Checkpointing with Message Logging beyond the Piecewise Deterministic (PWD) Model for Distributed Systems

Jinho Ahn †

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

Abstract: This paper introduces an effective communication-induced checkpointing protocol using message logging to enable the number of extra checkpoints to be far lower than the previous number. Even if a situation occurs in which it is decided that a process receiving a message has to perform forced checkpointing, our protocol allows the process to skip the forced checkpointing action if it recognizes that the state of its sender right before the receipt of the message is recoverable. Additionally, the communication-induced checkpointing protocol is thus not required to assume the piecewise deterministic model, despite being combined with message logging. This protocol can maintain these features by piggybacking a one-bit variable and an n-size vector on each message sent. Our simulation results verify our claim that the presented protocol performs much better than the representative optimized protocol with respect to the forced checkpointing frequency, regardless of the communication pattern.

Keywords: distributed systems; fault tolerance; checkpointing; message logging

check for updates

Citation: Ahn, J. Communication-Induced Checkpointing with Message Logging beyond the PWD Model for Distributed Systems. *Electronics* **2021**, *10*, 1428. https://doi.org/10.3390/electronics10121428

Academic Editors: Padmanabhan Balasubramanian and Costas Psychalinos

Received: 4 May 2021
Accepted: 11 June 2021
Published: 14 June 2021

1. Introduction

As parallel algorithms perform many operations on a cluster of independent computing nodes, even a single node crash can cause the execution of an algorithm to halt [1]. This undesirable property may make large-scale distributed systems more vulnerable to failures [2]. For this reason, effective fault-tolerance techniques are essentially needed in such systems. Rollback recovery is one such technique that enables the current erroneous state of a distributed system to be restored to a previous failure-free state from the stable storage [3,4]. To achieve this goal, the system recovery information must be occasionally saved to the storage for normal operations [5].

Among the techniques used for rollback recovery, checkpoint-based recovery depends solely upon local states of processes maintained in the stable storage—called checkpoints—to support fault tolerance [5]. In the case of a failure, the system state is recovered by using the most recent consistent global checkpoint kept in the storage. According to when and how consistent sets of checkpoints are formed, checkpoint-based recovery protocols are categorized into coordinated, independent, and communication-induced checkpointing [5]. In order to balance trade-offs between independent and coordinated checkpointing in an effective manner, communication-induced checkpointing (CIC) is used to preclude any local checkpoint that have already been taken from becoming useless by performing forced checkpointing while attempting to increase the degree of checkpointing independence as much as possible [6–15]. The CIC protocols include HMNR [8], which uses this feature to enable the number of extra checkpoints to be much lower by effectively using the control information contained in each sent message. Next, an improved HMNR protocol [12], LazyHMNR, attempts to use a lazy indexing strategy [13] to alleviate the problem of high frequencies of forced checkpointing that may occur in some particular cases. Next, two protocols, FINE [9] and LazyFINE [10], were designed to try to generate fewer forced checkpoints than HMNR and LazyHMNR with the same numbers of variables that they hold. However, they cannot ensure the property of never having useless checkpoints [7]. Then, an adaptive CIC protocol [11] was developed in an attempt to delay taking forced checkpointing actions as long as possible by evaluating some safety predicates. One of the most recent CIC protocols [14] utilizes the effectiveness of the one-to-many transmission of broadcasting links, which is widely used to lower the number of extra checkpoints. However, this may greatly degrade the applicability of the CIC. Lastly, several recent works [6,15] exploited one or more protocols mentioned above to raise the dependability of the systems to a higher level in the fields of distributed database management systems and web services.

Generally, checkpointing-based recovery, including CIC, is not subject to the piecewise deterministic (PWD) model; thus, it is less restrictive and more realistic for application in distributed systems than in message-logging-based recovery [16–18]. However, the former may not ensure the recovery of the system to its pre-failure state, which generally makes the rollback distance of each process much longer than that in the latter [5]. Thus, hybrid protocols that combine the two techniques were proposed to compensate for the drawbacks of the first [5]. However, these protocols cannot be freed from the PWD assumption. For this reason, none of the existing CIC protocols—including a family of HMNR protocols [6,8–15]—developed so far can utilize the benefits of message logging on real-world distributed systems without assuming the PWD model. Whenever each process receives a message, the protocols cause the process to perform forced checkpointing if they decide that one or more checkpoints that have already been taken may become useless. Due to this inherent shortcoming, even the HMNR protocols force each process to take extra checkpoints—more than twice the number of basic checkpoints [7]. However, the property of never having useless checkpoints is ensured without performing the forced checkpointing action if the process can know precisely that the current state of the sender of the message can be deterministically restored by replaying logged messages from the stable storage in the case of a failure of the sender. We found that this observation may be an important way to drive a large reduction in the number of forced extra checkpoints that all of the previous CIC protocols can incur. This paper introduces a CIC protocol, S-CIC, with message logging that is not subject to the PWD model in order to address the aforementioned observation. The protocol can achieve this goal by only carrying a one-bit variable and an n-size vector into every message transmitted. In the checkpointing and communication patterns where the existing CIC protocols—including the family of HMNR protocols—force the receiver of a message to take extra checkpoints, if the information of the message sender piggybacked on it indicates that the sender's state right before sending is recoverable, the proposed protocol allows the receiver not to perform forced checkpointing.

2. Background

2.1. Fundamentals

In this paper, we assume a distributed system with no global clock or memory and immunity to network partition [5]. Every process can crash according to the fail-stop failure model [19], and the processes collaborate with others only by making reliable message exchanges through an asynchronous transmission channel [5,8]. Each process p begins an execution from its first state and performs a combination of internal, message-sending, and message-delivering events [5]. Here, internal events are produced to perform their individual computations with no interactions with others. All of the event processes that are incurred for normal operations are sequenced according to Lamport's "happened before" relation [20].

Ck_p^i represents the ith local checkpoint of p, and $\mathrm{Ck}_p^i.lc$ is the local timestamp assigned to Ck_p^i when it is taken. Assume that each process p records the first checkpoint, Ck_p^0, on the storage containing its initial state when it begins its own computation. A global checkpoint means a set of local checkpoints that hold only one per process in the system [5,7]. A pair of local checkpoints (Ck$_p^i$,Ck$_q^j$) is named mutually consistent if and only if there is no case in which m is delivered before Ck$_q^j$, but is sent after Ck$_p^i$. A global checkpoint is consistent if and only if every couple of local checkpoints belonging to the first always satisfies the mutual consistency condition [21]. The concept of the Z-path [22] is exploited to check if the condition of mutual consistency is satisfied on an ordered pair of checkpoints by finding causal sub-paths, as well as non-causal (NC) sub-paths, where the two checkpoints can be connected to each other. A Z-path that includes a cycle from a local checkpoint Ck$_p^i$ to itself is called a Z-cycle [22].

Theorem 1. *For any pair of checkpoints Ck$_p^i$ and Ck$_q^j$, if a Z-path begins with Ck$_p^i$ and terminates with Ck$_q^j$ and Ck$_p^i$.lc is lower than Ck$_q^j$.lc, then no Z-cycle can form [22].*

2.2. Related Work

HMNR is an optimized protocol that aims to have no local checkpoints that are useless while decreasing the number of extra checkpoints. To keep this feature in HMNR, each process p should always have the following five state variables [8]: lc_p is a non-negative integer variable that has the present value of p's local timestamp. $send\_to_p$ is a vector in which $send\_to_p[q]$ keeps a boolean value to detect a non-causal path to q from p. $ckpt_p$ is a vector in which $ckpt_p[q]$ contains the total number of checkpoints that q has recorded in the stable storage from its initial execution that p currently recognizes. $taken_p$ is a vector where $taken_p[q]$ keeps the boolean value for q to indicate the existence of at least one causal Z-path from the latest checkpoint of q that p perceives to the subsequent checkpoint for p. $greater_p$ is a vector where $greater_p[q]$ has the boolean value for q, and this indicates whether p's current timestamp lc_p is greater than the most recent timestamp of q perceived by p (=true) or not (=false).

This protocol includes a checkpoint-timestamping mechanism that uses Lamport's logical clock [21] to satisfy Theorem 1, implying that increasing the timestamp flow along any Z-path always ensures that no checkpoint becomes useless. The mechanism is sufficient for ensuring that no causal Z-path includes a Z-cycle formation [8,22].

However, two kinds of non-causal (NC) Z-paths [8,22], as shown in Figure 1, can violate the theorem, even if the timestamping mechanism is used. To prevent Z-cycles from forming in these cases, HMNR forces each process p after receiving a message m to save an additional checkpoint if the following condition C_{HMNR} is satisfied.

- $C_{HMNR} \equiv C_1 \vee C_2$
- $C_1 \equiv \exists j(1 \leq j \leq n){:}sent\_to_p[j] \wedge m.greater[j] \wedge (m.lc > lc_p)$
- $C_2 \equiv (ckpt_p[p] = m.ckpt[p]) \wedge m.taken[p]$

The first case is an NC Z-path pattern connecting two checkpoints, Ck$_p^i$ and Ck$_r^{k+1}$, as shown in Figure 1a. In this example, three processes, p, q, and r, are exchanging messages: m_1, m_2, and m_3. As Ck$_p^i.lc$ = Ck$_r^{k+1}.lc$, the path violates the theorem. When q sends m_2 to r, $sent\_to_q[r]$ becomes true. However, p can get lc_r before Ck$_r^{k+1}$ through m_3, $m_3.lc(=lc_r) < lc_p$. Thus, $greater_p[r]$ still remains true. Then, it is brought to q when receiving m_1, so $m1.greater[r]$ = true and $m_1.lc(=lc_p) > lc_q$. As the first sub-condition of C_{HMNR}, C_1, is satisfied, HMNR forces q to record an extra checkpoint Ck$_q^{j+1}$ in the storage before conveying m_1 to the target application.

The second case is another NC Z-path pattern with Ck$_p^i$ and Ck$_r^{k+1}$, as shown in Figure 1b. In the figure, the path incurs a Z-cycle involving Ck$_r^{k+1}$ because r sends m_3 to p after Ck$_r^{k+1}$, and then q receives m_1, depending on m_3, from p. In this case, as m_3 is transmitted to p from r, $greater_p[r](\leftarrow greater_p[r] \wedge m_3.greater[r])$ becomes false as $m_3.lc(=lc_r)$ = lc_p and $m_3.greater[r]$ = false. As $m_1.greater[r]$ = false when q receives m_1, C_1 is not sufficient for detecting the violation. Therefore, the second sub-condition of C_{HMNR}, C_2, needs to be checked. When taking Ck$_r^{k+1}$, $taken_r[q]$ changes to true while the value of $ckpt_r[q]$ is still the same as the number of checkpoints associated with Ck$_q^j$. Then, the two values can be brought to q through a causal Z-path composed of m_3 and m_1 in order. As $m_1.ckpt[q]$ = $ckpt_q[q]$ and $m_1.taken[q]$ = true, C_2 is satisfied.

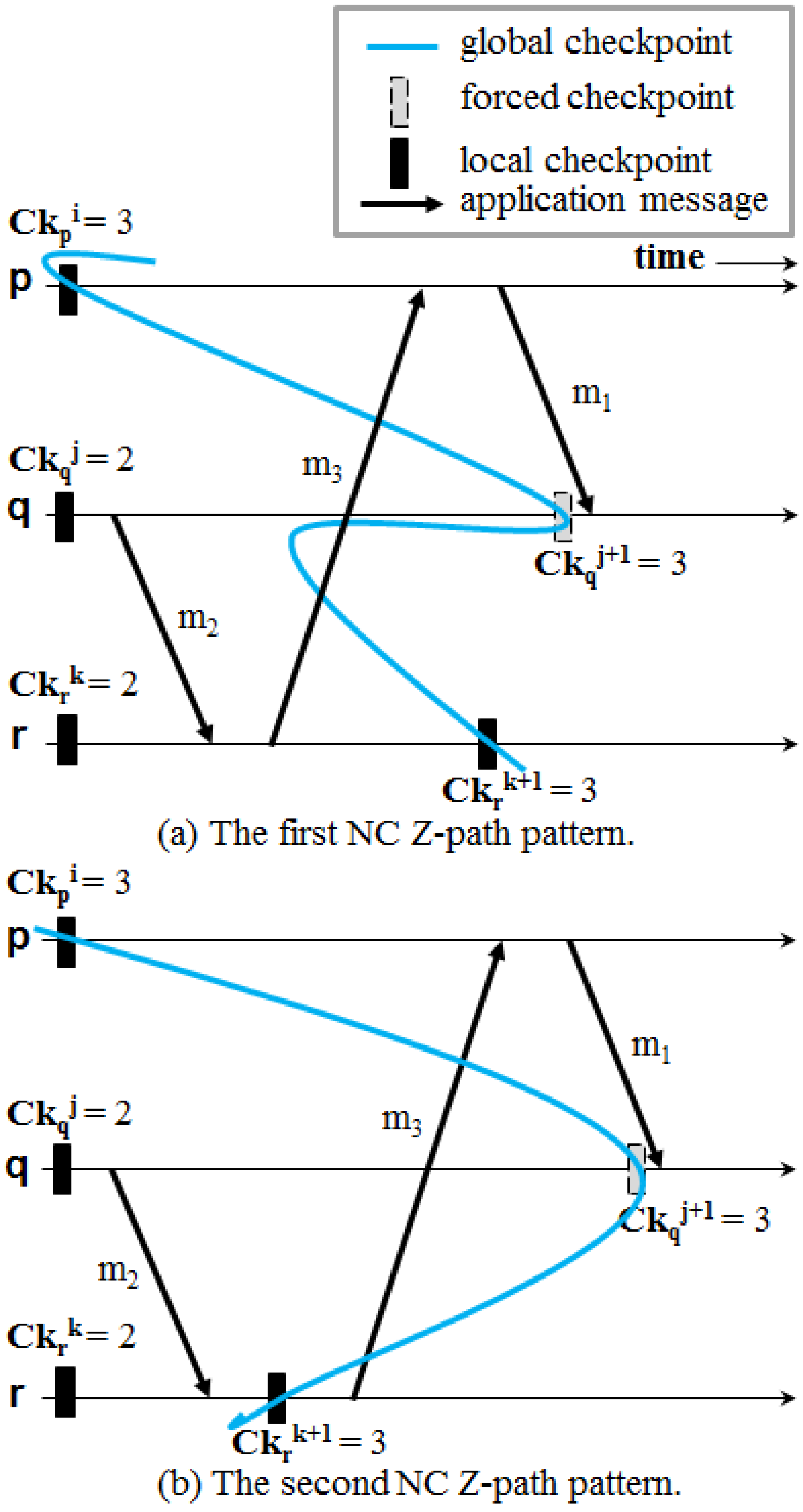

Figure 1. Preventing uselessness of local checkpoints in HMNR.

However, in both examples, if p's state right before sending m_1 is recoverable and q knows this, q does not need to take Ck_q^{j+1} before delivering m_1, even though C_{HMNR} is satisfied. In other words, when q recognizes that both m_2 and m_3 can always be replayed in case of failure and that p's internal execution before sending m_1 is deterministic, Ck_r^{k+1} will not be a useless checkpoint, even though Ck_q^{j+1} is taken after delivering m_1. Based on this new observation, we present a low-overhead CIC protocol, S-CIC, that uses message logging to detect this type of recoverability in an efficient manner without assuming the PWD constraint.

The enhanced version of HMNR [12], LazyHMNR, attempts to lessen the forced checkpointing frequency in unconventional cases that may take place due to asymmetries in the rates of increase of logical timestamps. It fulfills this requirement by delaying the swift growth of the logical timestamps of some processes—which is caused by repeated checkpointing actions on them—as long as possible.

Another CIC protocol [9], FINE, attempts to intensify the optimality of the consistency predicate of HMNR with only its mandatory state variables in order to ensure the property of never having useless checkpoints. The advanced version of FINE [10], LazyFINE, was designed to incorporate the laziness of logical timestamp increases into FINE. However, it was proved that the two protocols can create useless checkpoints because the Z-consistent timestamping rule cannot be enforced [7].

A delayed CIC protocol [11], DCFI, was introduced to lower the forced checkpointing frequency by applying several safety rules that enabled the postponement of checkpointing enforcements. This feature may have a much lower total number of checkpoints that are taken in the system. However, the protocol does not incorporate a method for significantly lowering the frequency of extra checkpointing actions by exploiting the rollback distance reduction benefit of message logging.

Another CIC protocol [14], BN-FI, was recently presented in order to curtail the number of extra checkpoints by exploiting the functional strength that broadcasting networks generally hold, which is called one-to-many transmission effectiveness. This special capability of lightweight group dissemination can speed up the updating of the last logical timestamp of each transmitter for the others on the network. This behavioral property enables each process to precisely detect if the ongoing Z-path has at least one checkpoint that has become useless much earlier than in the previous protocols. However, the performance gain of the protocol limits its applicability to network environments.

One of the most recent CIC algorithms [15] was developed in order to maintain a globally consistent state of each transaction in a distributed database management system while making the delay in failure-free transaction execution as short as possible. The algorithm attempts to enhance the recoverability of the system states with a far lower number of extra checkpoints by recording only the states of the completely committed transactions in the stable storage. However, the algorithm has the same shortcomings as those of the original HMNR protocol mentioned above.

The authors of [6] proposed an adaptive checkpoint generation algorithm in order to decrease the frequency of forced checkpointing actions by considering the system's behavior in comparison with the static algorithm, which did not reflect the environmental changes onto web services being operated. The algorithm made decisions on whether forced checkpoints should be taken based on the quality of the service parameters and the policies currently applied to their corresponding web services. In order to improve the dependability of the web services, three kinds of CIC protocols were exploited: HMNR [8], DCFI [11], and FINE [9]. Among them, HMNR and DCFI performed better than FINE on the system in terms of the number of forced checkpoints. However, the system still bore the respective limitations of the three CIC protocols stated above.

However, as all of the CIC protocols mentioned above attempt to use message logging to shorten the rollback distance of each process during recovery as much as they can, they must be applied only to deterministic services and systems, resulting in a large contraction of the scope of the application areas. Table 1 shows a summary of a comparison of some primary features of the CIC representatives.

Table 1. Comparison with the other CIC representatives (* non-deterministic).

Feature	S-CIC	HMNR	FINE	DCFI	BN-FI
No useless checkpoint existence	Yes	Yes	No	Yes	Yes
Broadcast network only	No	No	No	No	Yes
Model assumed if message logging used	ND *	PWD	PWD	PWD	PWD

3. The Proposed Protocol

S-CIC was devised to maintain the following three behavioral properties.

- Similarly to HMNR, each process attaches the state information related to other processes, as well as to itself, to every outgoing message so that the number of extra checkpoints decreases as much as possible.
- Even if either of the two cases where a process should perform a forced checkpointing before delivery of a message in HMNR occurs, S-CIC does not have the process perform the task if it knows that the same message can be replayed in spite of any future failures.
- Although pessimistic message logging is used to satisfy the second requirement, S-CIC is not subject to the PWD model.

Initially, each process deterministically performs its computation in a certain interval and, if a non-deterministic (ND) event occurs in this interval, the process begins its ND execution interval. In this research field, ND events can be classified into two types of events. The first type includes loggable ND events, of which there is sufficient support for forcing the replay at the same point in case of failure. Message receipt is one type of loggable ND event that most message-logging protocols detect and save in the stable storage for recovery. Aside from this, there are other types of loggable ND events, such as software interrupts or signals, which some other works [2] attempted to detect in order to make it possible to replay them in case of failure. This effort may raise the rate of the deterministic (DM) execution intervals. The second type comprises unloggable ND events, for which there is no support for taking action to enable a form of repeatable execution in case of failure.

To hold all of them in S-CIC, each process p should always have the following additional state information:

- $SSNmV_p$: A vector that saves an element composed of two variables, ssn and ND, for each process q. $SSNmV_p[q].ssn$ keeps the value of the ssn of the latest message m that q has transmitted and is known by p. $SSNmV_p[q].ND$ is a boolean value that indicates whether at least one internal unloggable ND event q has been executed before m since q's latest checkpoint. The two variables are initialized to (0,false). As a message m is transmitted from p, $SSNmV_p[p].ssn$ increments by one and the vector is attached to m. If p takes a local checkpoint, $SSNmV_p[p].ND$ comes back to false. If a message m is transmitted to p, $SSNmV_p[q]$ is updated to $m.SSNmV[q]$ if $m.SSNmV[q].ssn > SSNmV_p[q].ssn$.
- $ND\text{-}mode_p$: A boolean variable for detecting whether it there is at least one internal unloggable ND event that any process including p has executed since its latest checkpoint. It is initialized to false and, when a message m is transmitted, it is attached to m. As p performs the receipt of a message or takes a checkpoint, $ND\text{-}mode_p$ comes back to false if $ND\text{-}mode_p$ = true and $\forall q(1 \leq q \leq n)$: $SSNmV_p[q].ND$ = false.
- rsn_p: A non-negative integer variable that has the same sequence number as that of the most recent message that p has received.

First, we aimed to understand how to identify the recoverable state of each process with checkpointing and message logging without assuming the use of the PWD model.

For this purpose, an execution mode detection method was introduced in order to consider three typical cases that can occur in CIC, as shown in Figure 2. As shown in Figure 2a, the first case is that a process performs its computation without any communication with others. In this example, a process p first executes with its internal DM or loggable ND events in a certain interval from its checkpointed state $\text{Ck}_p^i(SSNmV_p[p].ND$ = false), which is called the DM mode ($ND\text{-}mode_p$ = false). Then, if any first internal unloggable ND event occurs ($SSNmV_p[p].ND$ = true), p's ND interval begins and changes its execution mode to non-deterministic ($ND\text{-}mode_p$ = true). When taking its next checkpoint Ck_p^{i+1}, p's state becomes recoverable ($SSNmV_p[p].ND$ = false); thus, its execution mode returns to being deterministic ($ND\text{-}mode_p$ = false). Then, it performs its computation in a similar way. Therefore, in this case, if p fails at a certain execution point after Ck_p^{i+1}, it can restart from the latest checkpoint and recover to the state right before any first unloggable ND event after Ck_p^{i+1} without considering any dependency relation with others.

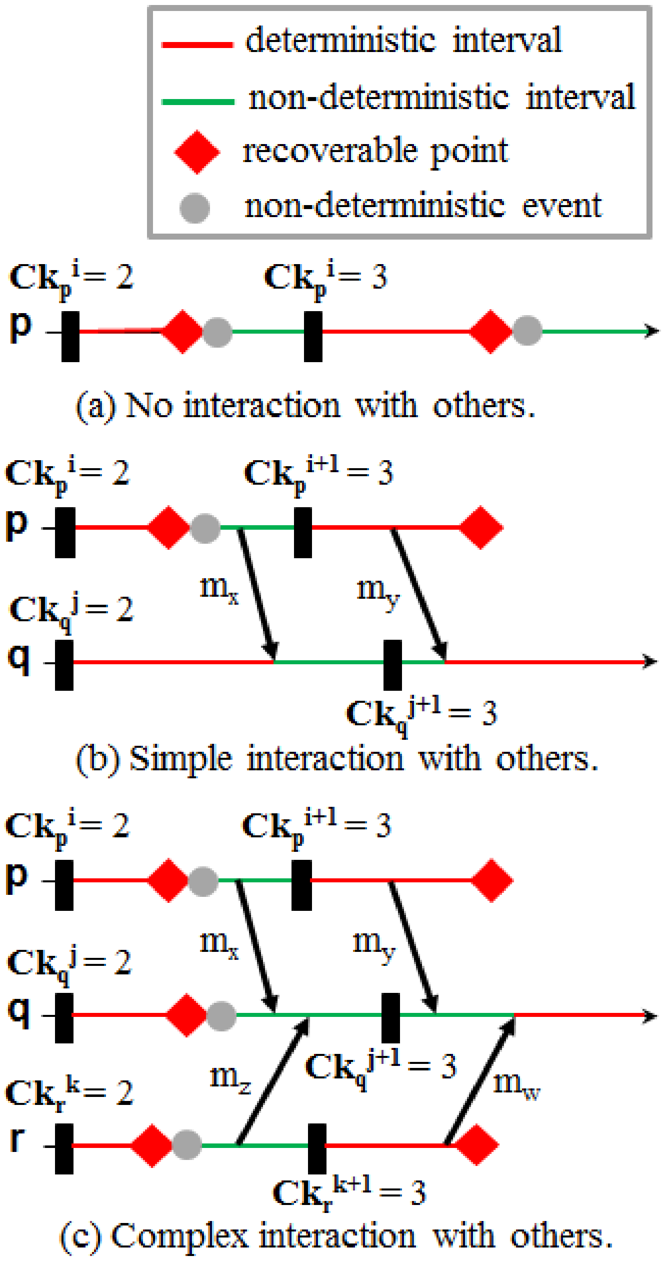

Figure 2. Three cases where the process execution mode changes.

As shown in Figure 2b, the second case is that an execution of a process q that is affected by messages transmitted from another single process. In this case, q first executes in its DM mode from its checkpointed state Ck_q^j($SSNmV_q[q].ND$ = false,$ND\text{-}mode_q$ = false). When receiving a message m_x from p, whose current mode is ND ($m_x.SSNmV[p].ND$ = true, $m_x.ND\text{-}mode$ = true), q's execution mode also changes to ND ($ND\text{-}mode_q$ = true). Then, q can execute with its internal unloggable ND events, although this is not shown in this figure ($SSNmV_q[q].ND$ = true). However, even if q takes its local checkpoint Ck_q^{j+1} ($SSNmV_q[q].ND$ = false), its execution mode still remains ND because q does not know that p's current state is recoverable ($SSNmV_p[p].ND$ = false, $ND\text{-}mode_p$ = false). When receiving m_y from p, q can recognize that p's current mode is DM due to the information piggybacked on m_y($m_y.SSNmV[p].ND$ = false, $m_y.ND\text{-}mode$ = false). Then, q's mode becomes DM($ND\text{-}mode_q$ = false).

As shown in Figure 2c, the third case is that a process q fulfills its computation depending on messages received from more than one process, p and r, which execute with internal unloggable ND events. In this case, q first executes in its DM mode from its checkpointed state Ck_q^j, and then in its ND mode with an unloggable ND event ($SSNmV_q[q].ND$ = true, $ND\text{-}mode_q$ = true). Next, it receives two messages, m_x and m_z, from p and r, which are currently both in ND mode ($m_z.SSNmV[r].ND$ = true,$m_z.ND\text{-}mode$ = true). When taking its next checkpoint Ck_q^{j+1}($SSNmV_q[q].ND$ = false), it still executes in ND mode even if the current modes of the two processes are both DM after their latest checkpoints. This unawareness can be resolved after q has received m_y and m_w($m_w.SSNmV[r].ND$ = false, $m_w.ND\text{-}mode$ = false).

Let us examine how, by using the mode detection method, S-CIC can have its number of extra checkpoints lowered in the two types of NC Z-path patterns in comparison with HMNR, as shown in Figure 3. An example of the first pattern of NC paths is illustrated in Figure 3a, which violates the theorem in HMNR. Let the ssns of p, q, and r be α, β, and γ for Ck_p^i, Ck_q^j, and Ck_r^k, respectively. When q sends m_2 to r, its mode is DM($ND\text{-}mode_q$ = false) and its ssn is (β + 1)($SSNmV_q[q]$ = (β + 1,false)). The two pieces of information are piggybacked on m_2. On receiving m_2, r is in the DM mode ($SSNmV_r[r]$ = (γ,false),$ND\text{-}mode_r$ = false). Then, it increments its rsn, rsn_r, and saves a log element whose form is e (sender's identifier, receiver's identifier, ssn, rsn, data) in the stable storage—for example, e (q, r, (β + 1), rsn_r, $m_2.data$) for m_2. Afterwards, it updates its mode-detection-related information as follows: $ND\text{-}mode_r \leftarrow m_2.ND\text{-}mode \vee ND\text{-}mode_r$ = false, $SSNmV_r$ = $\{$(0,false),(β + 1,false), (γ,false)$\}$. After receiving m_3, whose ssn is (γ + 1) from r, and logging it, p's mode remains DM, as r and p are both in DM mode ($ND\text{-}mode_p \leftarrow m_3.ND\text{-}mode \vee ND\text{-}mode_p$ = false), and its ssn vector $SSNmV_p$ is updated to $\{$(α,false), (β + 1,false),(γ + 1,false)$\}$. When q receives m_1 from p and it is logged, it is in ND mode ($SSNmV_q[q]$=(β + 1,true),$ND\text{-}mode_q$ = true), and C_{HMNR} becomes true, as $sent\_to_q[r]$, $m.greater[r]$, and $m.lc > lc_q$ are all true. However, q does not need to take any forced checkpoints ($m_1.ND\text{-}mode \wedge C_{HMNR}$ = false) because it knows p's state, including the fact that sending m_1 is recoverable ($m_1.ND\text{-}mode$ = false); thus, m_1 can always be regenerated even if p crashes. When taking its next checkpoint Ck_q^{j+1}, q's mode changes to DM and its vector $SSNmV_q$ is updated to $\{$(α + 1,false),(β + 1,false),(γ + 1,false)$\}$. Therefore, in this example, p's recoverable state until sending m_1 after Ck_p^i, Ck_q^{j+1}, and Ck_r^{k+1} comprises a globally consistent state.

Figure 3b illustrates an example of the second pattern of NC paths that violate the theorem in HMNR. After r receives m_2 from q, whose mode is DM ($m_2.ND\text{-}mode$ = false,$m_2.SSNmV$ = $\{$(0,false),(β + 1, false),(0,false)$\}$), and logging it, its mode is ND($m_2.ND\text{-}mode \vee ND\text{-}mode_r$ = true) because of its internal unloggable ND event ($SSNmV_r[r]$ =(γ,true), $ND\text{-}mode_r$ = true). However, when taking a checkpoint Ck_r^{k+1}($SSNmV_r[r]$= (γ,false)), r's mode changes to DM; thus, m_3 can always be replayed ($m_3.ND\text{-}mode$ = false). On m_3's receipt and logging, p keeps its DM mode and updates its variables as follows: $ND\text{-}mode_p \leftarrow m_3.ND\text{-}mode \vee ND\text{-}mode_p$ = false, and $SSNmV_p$ = $\{$(α,false),(β + 1,false),(γ + 1, false)$\}$. When q receives m_1 from p and logs it, C_{HMNR} becomes true, as $m_1.ckpt[q]$=$ckpt_q[q]$

and $m_1.taken[q]$ are all true. However, q does not need to take any forced checkpoints ($m_1.ND\text{-}mode \wedge C_{HMNR}$ = false) because r and p can deterministically reproduce and send m_3 and m_1 in order, respectively, even in case of p's failure ($m_1.ND\text{-}mode$ = false). Therefore, Ck_q^{j+1}, which is taken after delivering m_1($ND\text{-}mode_q$ = false, $SSNmV_q[q]$=(β + 1,false)), is consistent with the states of the others right after sending m_1 and m_3.

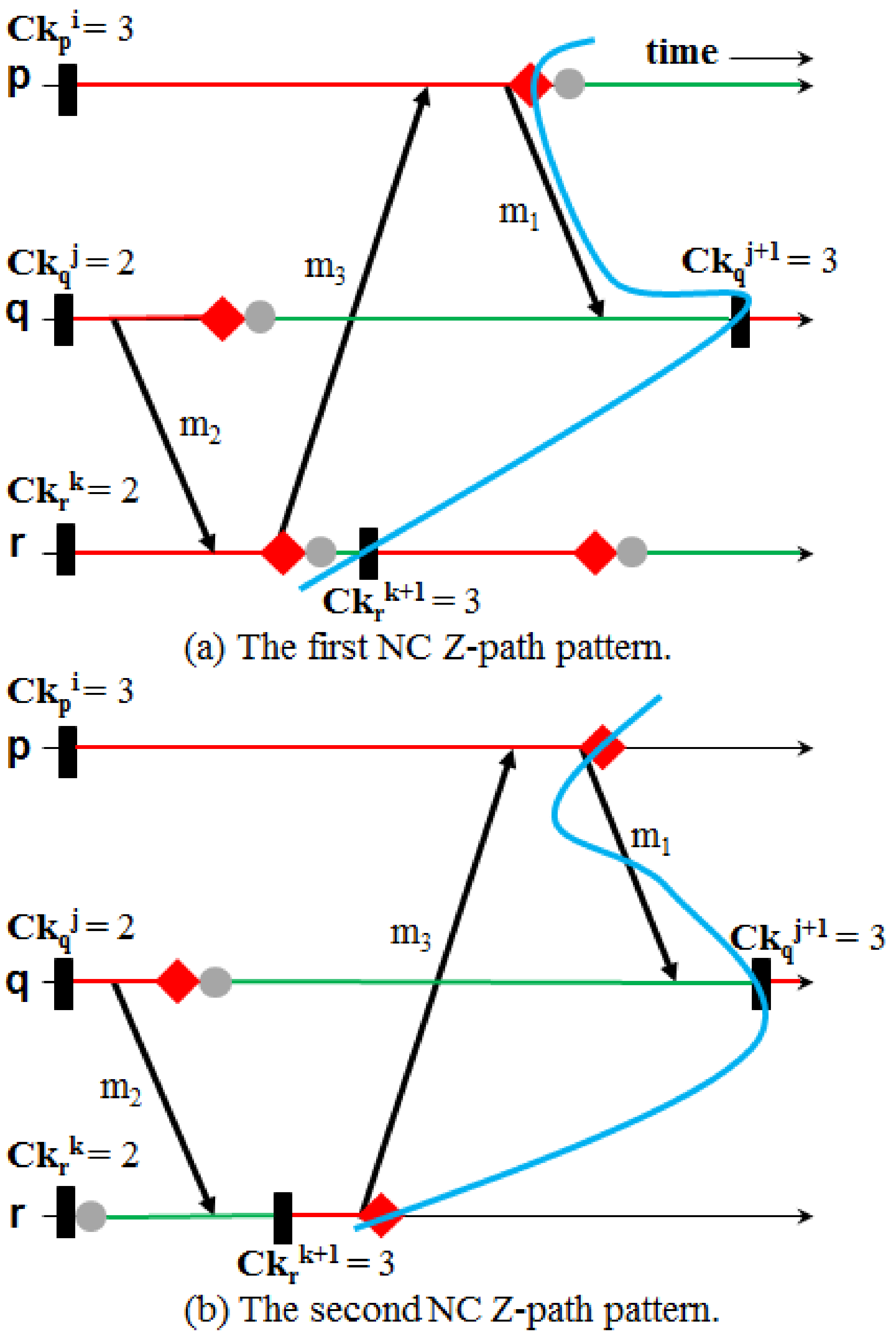

Figure 3. How S-CIC avoids forced checkpointing, unlike HMNR.

Figure 4 presents the concrete algorithmic description of S-CIC.

```
Module INITIALIZE()
    $lc_p \leftarrow 0$; $rsn_p \leftarrow 0$; $ND\text{-}mode_p \leftarrow$ false;
    $taken_p[p] \leftarrow$ false; $greater_p[p] \leftarrow$ false;
    $\forall r(1 \le r \le n)$: $ckpt_p[r] \leftarrow 0$; $SSNmV_p[r].ssn \leftarrow 0$;
                $SSNmV_p[r].mode \leftarrow$ false;
 // Take the first checkpoint of $p$. //
    invoke Module Local-Checkpointing();
    return;   // The end of this module. //

Module TRANSMIT-MSG($receiver$,$data$)
    if($send\_to_p[receiver]$ = false) then $send\_to_p[receiver] \leftarrow true$;
    increment $SSNmV_p[p].ssn$ by one;
    send $m(lc_p, ND\text{-}mode_p, SSNmV_p, greater_p, ckpt_p, taken_p, data)$ to $receiver$;
    return;   // The end of this module. //

Module RECV-MSG($m(lc, ND\text{-}mode, SSNmV, greater, ckpt, taken, data)$) FROM $s$
    if($m.SSNmV[s].ssn > SSNmV_p[s].ssn$) then
 // Update mode information of the other processes. //
      for all $r(1 \le r \le n)$ st $r \ne p$ do
          if($m.SSNmV[r].ssn > SSNmV_p[r].ssn$) then
            $SSNmV_p[r].ssn \leftarrow m.SSNmV[r].ssn$;
            $SSNmV_p[r].mode \leftarrow m.SSNmV[r].mode$;
 // Check if the state of $s$ on sending $m$ is recoverable. //
      if($ND\text{-}mode_p$ = true $\wedge$ $m.ND\text{-}mode$ = false) then
        if($\forall r(1 \le r \le n, r \ne p)$: $SSNmV_p[r].mode$ = false) then
          $ND\text{-}mode_p \leftarrow$ false;
 // Check if $p$ has to perform a forced checkpointing before delivering $m$. //
      $C_{HMNR} \leftarrow (\exists r(1 \le r \le n)$: $send\_to_p[r] \wedge m.greater[r]$
                $\wedge (m.lc > lc_p)) \vee ((ckpt_p[p] = m.ckpt[p]) \wedge m.taken[p])$;
      $FCTaken \leftarrow m.ND\text{-}mode \wedge C_{HMNR}$;
      $ND\text{-}mode_p \leftarrow ND\text{-}mode_p \vee m.ND\text{-}mode$;
      if($FCTaken = true$) then
        invoke Module Local-Checkpointing();
      increment $rsn_p$ by one ;
      save a log element $e(s, p, m.SSNmV[s].ssn, rsn_p, m.data)$
            on the stable storage;
      have $m.data$ delivered to the destination application;
 // Update its local timestamp and Z-cycle detection variables. //
      if($m.lc > lc_p$) then
        $greater_p[p] \leftarrow$ false; $lc_p \leftarrow m.lc$;
        $\forall r(1 \le r \le n, r \ne p)$: $greater_p[r] \leftarrow m.greater[r]$;
      else if($m.lc = lc_p$) then
           $\forall r(1 \le r \le n)$: $greater_p[r] \leftarrow greater_p[r] \wedge m.greater[r]$;
      for all $r(1 \le r \le n)$ st $r \ne p$ do
          if($m.ckpt[r] > ckpt_p[r]$) then
            $ckpt_p[r] \leftarrow m.ckpt[r]$; $taken_p[r] \leftarrow m.taken[r]$;
          else if($m.ckpt[r] = ckpt_p[r]$) then
             $taken_p[r] \leftarrow taken_p[r] \wedge m.taken[r]$;
    return;   // The end of this module. //
```

Figure 4. *Cont.*

```
Module EXEC-NDEVENT(event)
    if(there is no support for forcing the replay of event
        at the same point in case of failure) then
        $ND\text{-}mode_p \leftarrow$ true; $SSNmV_p[p].mode \leftarrow$ true;
    else
        take action to enable a form of repeatable execution of event;
    return;   // The end of this module. //

Module LOCAL-CHECKPOINTING()
    increment $lc_p$ and $ckpt_p[p]$ by one respectively;
    $SSNmV_p[p].mode \leftarrow$ false;
    $\forall r(1 \leq r \leq n)$: $send\_to_p[r] \leftarrow$ false;
    $\forall r(1 \leq r \leq n, r \neq p)$: $taken_p[r] \leftarrow$ true; $greater_p[r] \leftarrow$ true;
    if($ND\text{-}mode_p$ = true $\wedge$ $\forall r(1 \leq r \leq n)$: $SSNmV_p[r].mode$ = false) then
        $ND\text{-}mode_p \leftarrow$ false;
    take its local checkpoint with ($lc_p$,$rsn_p$,$ND\text{-}mode_p$, $SSNmV_p$,$ckpt_p$);
    return;   // The end of this module. //
```

Figure 4. Modules of p for S-CIC.

Definition 1. *$Z_{p_i}^{r_{k+1}}$ is an arbitrary Z-path from Ck_p^i to Ck_r^{k+1}.*

Definition 2. *$NC\text{-}Z\text{-}PATH(Z_{p_i}^{r_{k+1}})$ is a variable-length non-causal sub-Z-path of $Z_{p_i}^{r_{k+1}}$.*

Lemma 1. *If, in $NC\text{-}Z\text{-}PATH(Z_{p_i}^{r_{k+1}})$, $Ck_p^i.lc \not< Ck_r^{k+1}$, but p's state immediately before sending m to another process $q(q \neq r)$ is recoverable, no forced checkpointing is required to ensure that no useless local checkpoints exist.*

Proof. The correctness of the theorem is proved through contradiction. Assume that when m arrives at q, q has to perform a forced checkpoint action to enforce the safety condition, as q recognizes $Ck_p^i.lc \not< Ck_r^{k+1}$. Afterwards, if p crashes, it can recover to hold up to the state right after sending m. The state is always consistent with q's state immediately after receiving m. Thus, the forced checkpoint is not needed to preclude Ck_r^{k+1} from being useless.

Therefore, if $Ck_p^i.lc \not< Ck_r^{k+1}$, but p's state immediately before sending m is recoverable, the property of not having any useless checkpoints can be kept without q's forced checkpointing before delivering m. This contradicts the hypothesis. □

Theorem 2. *S-CIC ensures that no checkpoint is useless.*

Proof. S-CIC includes a checkpoint-timestamping mechanism that uses Lamport's logical clock, as in HMNR. This feature ensures that any variable-length causal sub-Z-path in $Z_{p_i}^{r_{k+1}}$ includes no Z-cycles. Therefore, we only have to prove that the protocol prevents $NC\text{-}Z\text{-}PATH(Z_{p_i}^{r_{k+1}})$ from containing any Z-cycle formations. The proof goes on by induction on its length, denoted by $LENGTH(NC\text{-}Z\text{-}PATH(Z_{p_i}^{r_{k+1}}))$.
[Base case] In this case, $Z_{p_i}^{r_{k+1}}$ is an NC Z-path with two messages, m_1 and m_2, where Ck_p^i is the latest checkpoint before transmitting m_1 from p and Ck_r^{k+1} is the most recent checkpoint after m_2 is received by r. Suppose that q takes a checkpoint Ck_q^j, sends m_2 to r, and then receives m_1 from p before its next checkpoint. At this point, two cases must be checked.
Case 1: $m_1.ND\text{-}mode$ = false.
q knows p's state, including that sending m_1 is recoverable. Based on Lemma 1, even if q takes no forced checkpoints, Ck_r^{k+1} is always useful.
Case 2: $m_1.ND\text{-}mode$ = true
If $m_1.lc \leq m_2.lc$, $Ck_p^i.lc \leq m_1.lc \leq m_2.lc < Ck_r^{k+1}$. Thus, $Z_{p_i}^{r_{k+1}}$ includes no Z-cycles. Otherwise, two sub-cases must be checked.
Case 2.1: No causal sub-paths are generated from r to p or q between after r receives m_2

and before p sends m_1 or q receives it.
In this case, before p sends m_1 to q, $greater_p[r]$ is true because p knows that $lc_p > lc_r$. When q receives m_1, it has to perform a forced checkpointing action before delivery of m_1 because C_1 is satisfied.
Case 2.2: A causal sub-path, u, from r to p or q exists between after r received m_2 and before p sends m_1 or q receives it.
At this point, two sub-cases must be checked.
Case 2.2.1: u occurs at r before Ck_r^{k+1}.
If $lc_p \leq u.lc$, $Ck_p^i.lc < Ck_r^{k+1}.lc$ and Ck_r^{k+1} never becomes a useless checkpoint, regardless of whether u's destination is p or q. If p receives u, it updates $greater_p[r]$(=$m_1.greater[r]$) as false. If u goes to q, lc_q is updated with $u.lc$ and is greater than or equal to $m_1.lc$. In both cases, S-CIC can recognize that C_1 is not satisfied, and q does not perform a forced checkpointing action.
Otherwise, $Ck_p^i.lc \not< Ck_r^{k+1}$ and Ck_r^{k+1} may become useless in both cases. If p receives u, this condition causes $greater_p[r]$(=$m_1.greater[r]$) to remain unchanged (true). In addition, when m_1 is transmitted to q, $m_1.lc > lc_q$, as $u.lc \geq lc_q$. If u goes to q, $m_1.greater[r]$ is true and $m_1.lc > lc_q$. In both cases, S-CIC can recognize that C_1 is satisfied and causes q to perform a forced checkpointing action before delivering m.
Case 2.2.2: u occurs at r after Ck_r^{k+1}.
On receiving m_2, r can keep the value of the latest checkpoint index of q in $ckpt_r[q]$. As it takes Ck_r^{k+1}, $taken_r[q]$ is set to true. When r sends u, $ckpt_r[q]$ and $taken_r[q]$ are eventually brought to q by a directed path—either <u> or <u, m_1>. In both cases, S-CIC can recognize C_2 is satisfied and causes q to perform a forced checkpointing action before the delivery of the message received by q, as in HMNR.
[Induction hypothesis] It is assumed that the theorem is true for $Z_{p_i}^{r_{k+1}}$ if $LENGTH(NC\text{-}Z\text{-}PATH(Z_{p_i}^{r_{k+1}}))$ = l.
[Induction step] By the induction hypothesis, every checkpoint in $NC\text{-}Z\text{-}PATH(Z_{p_i}^{r_{k+1}})$ incurs no Z-cycle formations. Therefore, if a new non-causal path forms with $NC\text{-}Z\text{-}PATH(Z_{p_i}^{r_{k+1}})$, m_z (which is considered as one message in the induction hypothesis), and the (l + 1)-th message m_{l+1} together, the theorem is true for $Z_{p_i}^{r_{k+1}}$ if $LENGTH(NC\text{-}Z\text{-}PATH(Z_{p_i}^{r_{k+1}}))$ = l + 1. The following case is similar to the base case that was mentioned earlier.

Through induction, our protocol ensures that no checkpoint becomes useless. □

4. Performance Evaluation

Let us examine our extensive simulations to make a comparison of the performance of the two protocols, LazyHMNR and S-CIC, with a discrete-event simulation language, PARSEC [23]. LazyHMNR is one of the most recently developed versions of HMNR, which is intended to decrease the high frequency of forced checkpointing [7]. S-CIC is our improved version of LazyHMNR with the advantageous features mentioned in the previous sections.

In this comparison, we precisely examine one important performance index, $NOFC$. The index indicates the total number of forced checkpoints taken in each protocol. The simulated system is a cluster of N computers on a broadcast network. All processes running on each computer begin and finish their individual execution together. As a process transmits an application message, the message is destined to a single recipient at all times. The link capacity and propagation delay of the simulated network are 100 Mbps and 1 ms, respectively. Every process performs a basic local checkpointing task in a certain checkpoint interval according to an exponential distribution with a mean CI_{bc} = 5 min. In addition, among N processes, one is selected at random and transfers a message in every timed interval according to an exponential distribution with a mean of TI_{send} = 3 s. Furthermore, to measure the communication pattern sensitivity of the two protocols, more complex experiments were conducted by splitting applications into four groups: serial, circular, hierarchical, and irregular [24].

Figures 5–8 show the *NOFC* for both LazyHMNR and S-CIC with changes in the numbers of processes—denoted by *NOP*—scaling from 6 to 12 when the percentage of internal unloggable ND events in each process (*UND*) was 20%, 40%, 60%, and 80% for the four different communication patterns, respectively. In these figures, *UND* never changed the *NOFC* of LazyHMNR because, unlike S-CIC, LazyHMNR has no method for skipping forced checkpointing actions if the state of the sender of each message right before the receipt of the message is recoverable. As *NOP* increased, the ratio of the *NOFC* of LazyHMNR to that of S-CIC increased for all four patterns according to the change in *UND*, which ranged from 1.3 to 6.5. The main reason is that there was an increased possibility of forming the two kinds of NC Z-path patterns and inducing forced checkpointing in LazyHMNR when C_1 or C_2 was satisfied. In addition, the occurrence of fewer unloggable ND events per process (i.e., decreasing *UND*) led to a significant decrease in the forced checkpointing overhead of LazyHMNR due to the advantageous features of S-CIC. These results indicate that S-CIC frequently skips actions to take forced checkpoints for each process by checking the recoverability of the dependent states of other processes, unlike LazyHMNR. In addition, the features have the effect of a large reduction in the number of forced checkpoints, regardless of the communication patterns. However, the degree of their effectiveness may fluctuate in the irregular pattern because its irregularity can cause the formation of Z-cycles and can cause the first type of NC Z-path to differ in every run. From the results, we can see that, with these features, S-CIC can alleviate the shortcomings of the family of HMNR protocols, including LazyHMNR.

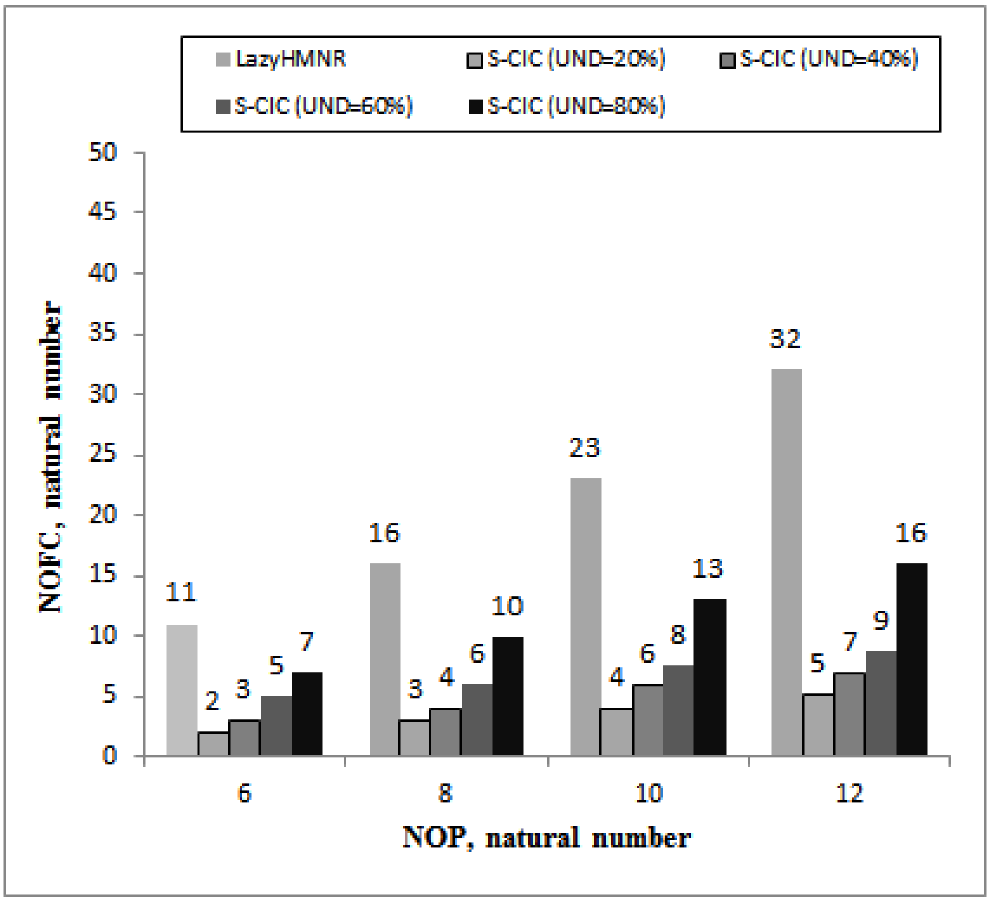

Figure 5. *NOFC* for the serial pattern.

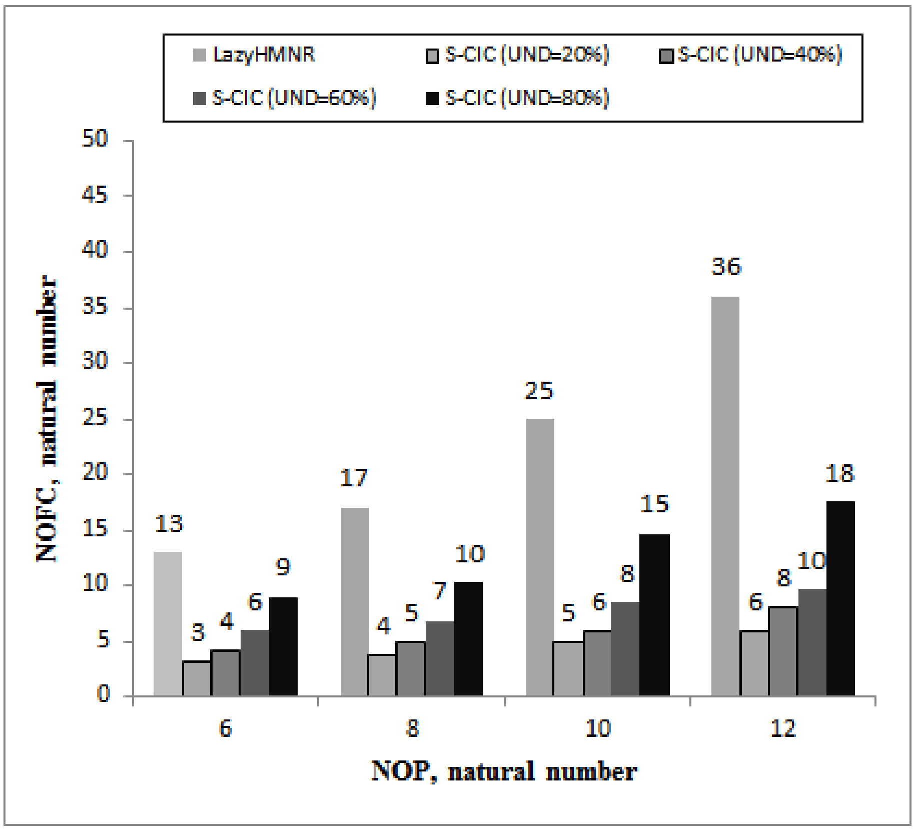

Figure 6. *NOFC* for the circular pattern.

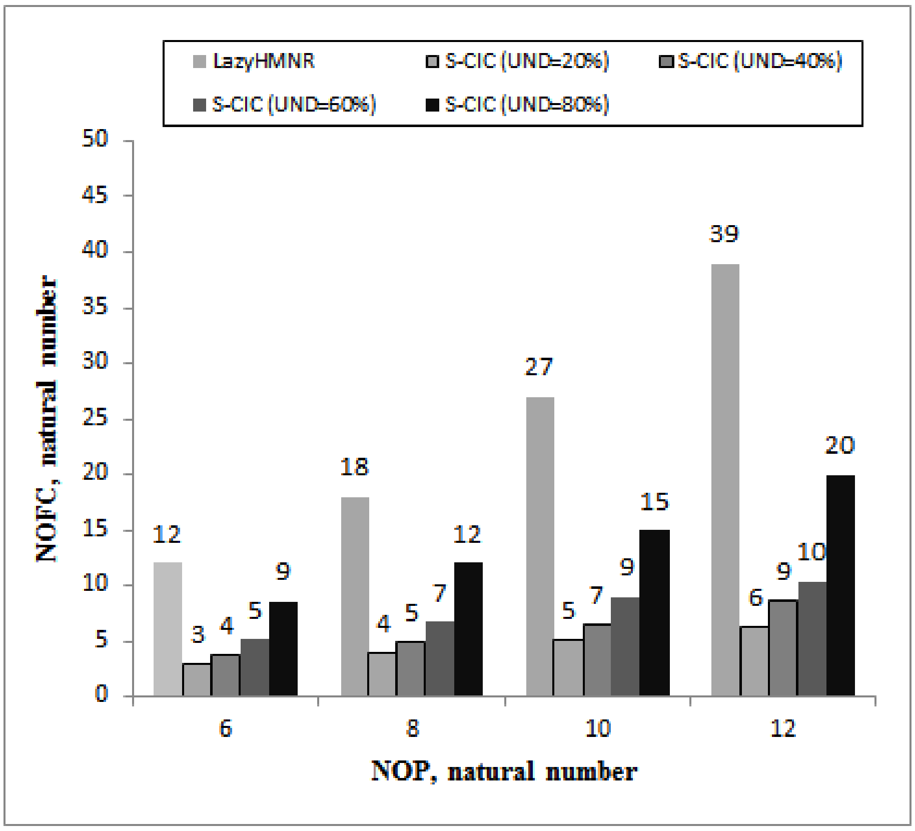

Figure 7. *NOFC* for the hierarchical pattern.

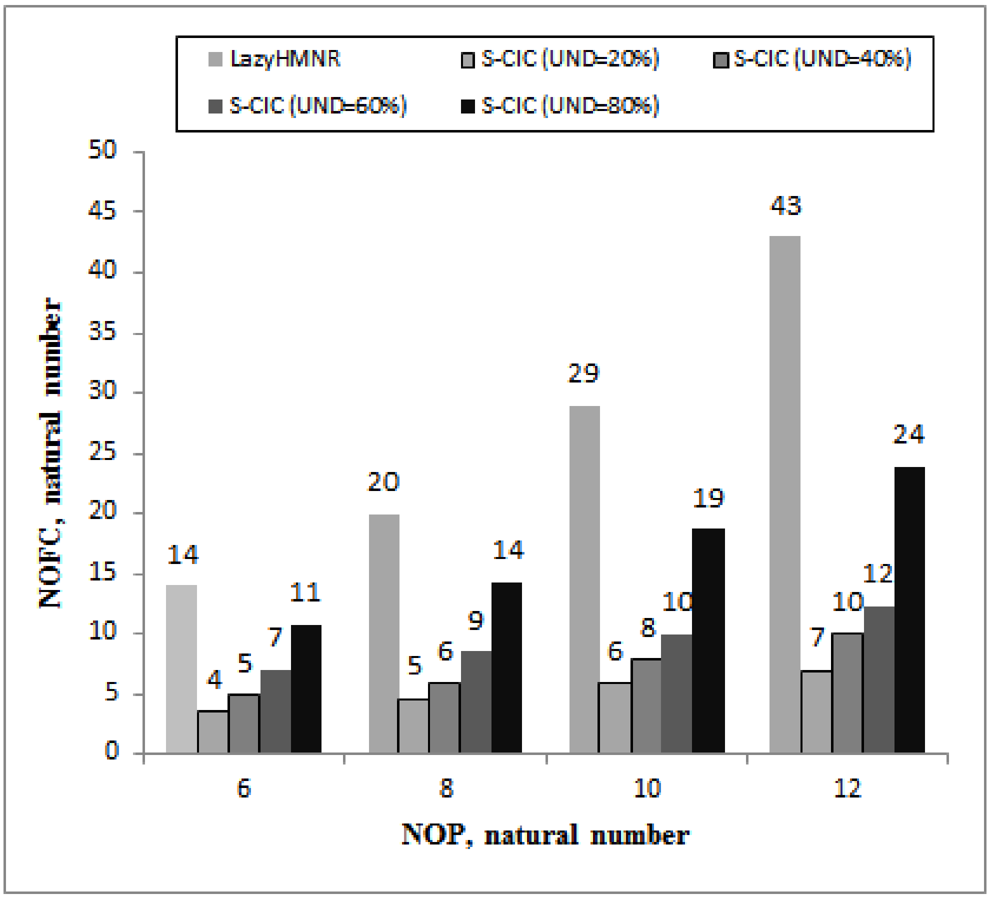

Figure 8. *NOFC* for the irregular pattern.

5. Conclusions

The proposed protocol, S-CIC, was developed in order to incorporate the following advantageous features. First, though situations can occur in which HMNR or LazyHMNR decides that, on receiving a message, a process has to perform forced checkpointing, S-CIC does not cause the process to perform this action when it recognizes that the state of its sender right before the receipt of the message is recoverable, leading to large reduction in the number of forced checkpoints compared with the family of HMNR protocols. Therefore, S-CIC is also not required to assume the PWD model, despite being combined with message logging. This goal can be realized by piggybacking the sender's recoverability status and a vector containing the last send sequence number and unloggable event occurrence status of every process onto each sent message. Our simulation results verified that the protocol outperforms the representative optimized protocol, LazyHMNR, with respect to the forced checkpointing frequency, regardless of the communication pattern used.

Funding: This research was funded by Kyonggi University, grant number 2018-033.

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. Nakamura, J.; Kim, Y.; Katayama, Y.; Masuzawa, T. A cooperative partial snapshot algorithm for checkpoint-rollback recovery of large-scale and dynamic distributed systems and experimental evaluations. *arXiv* **2021**, arXiv:2103.15285v1.
2. 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.

3. Estahbanati, M.G.; Schintke, F. Multilevel checkpoint/restart for large computational jobs on distributed computing resources. In Proceedings of the 38th Symposium on Reliable Distributed Systems (SRDS), Lyon, France, 1–4 October 2019; pp. 143–152.
4. Mansouri, H.; Pathan, A. Checkpointing distributed computing systems: An optimisation approach. *Int. J. High Perform. Comput. Appl.* **2019**, *15*, 202–209.
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. Vargas-Santiago, M.; Morales-Rosales, L.; Monroy, R.; Pomares-Hernandez, S.; Drira, K. Autonomic web services based on different adaptive quasi-asynchronous checkpointing techniques. *Appl. Sci.* **2020**, *10*, 2495. [CrossRef]
7. Garcia, I.C.; Vieira, G.M.D.; Buzato, L.E. A rollback in the history of communication-induced checkpointing. *arXiv* **2019**, arXiv:1702.06167v2.
8. Helary, J.-M.; Mostefaoui, A.; Netzer, R.H.B.; Raynal, M. Communication-based prevention of useless checkpoints in distributed computations. *Distrib. Comput.* **2000**, *13*, 29–43. [CrossRef]
9. Luo, Y.; Manivannan, D. FINE: A Fully Informed aNd Efficient communication-induced checkpointing protocol for distributed systems. *J. Parallel Distrib. Comput.* **2009**, *69*, 153–167. [CrossRef]
10. Luo, Y.; Manivannan, D. Theoretical and experimental evaluation of communication-induced checkpointing protocols in F_E and F_{Lazy-E}. *Perform. Eval.* **2011**, *68*, 429–445. [CrossRef]
11. Simón, A.; Hernandez, S.; Cruz, J.; Halima, R.; Kacem, H. Self-healing in autonomic distributed systems based on delayed communication-induced checkpointing. *Int. J. Auton. Adapt. Comm. Syst.* **2016**, *9*, 183–200. [CrossRef]
12. Tsai, J. Applying the fully-informed checkpointing protocol to the lazy indexing strategy. *J. Inf. Sci. Eng.* **2007**, *23*, 1611–1621.
13. Vieira, G.M.; Garcia, I.C.; Buzato, L.E. Systematic analysis of index-based checkpointing algorithms using simulation. In Proceedings of the IX Brazilian Symposium on Fault-Tolerant Computing, Florianapolis, Brazil, 5–7 March 2001; pp. 31–42.
14. Ahn, J. Efficient communication induced checkpointing protocol for broadcast network-based distributed systems. *Parallel Process. Lett.* **2019**, *29*, 1–12. [CrossRef]
15. Mansouri, H.; Pathan, A.S.K. A communication-induced checkpointing algorithm for consistent-transaction in distributed database systems. In Proceedings of the International Symposium on Security in Computing and Communication, Chennai, India, 14–17 October 2020; pp. 21–32.
16. Aung, T.; Min, Y.H.; Maw, A.H. Enhancement of fault tolerance in Kafka pipeline architecture. In Proceedings of the 11th International Conference on Advances in Information Technology, Bangkok, Thailand, 1–3 July 2020; pp. 1–8.
17. Ahn, J. Scalable sender-based message logging protocol with little communication overhead for distributed systems. *Parallel Process. Lett.* **2019**, *29*, 1–10. [CrossRef]
18. 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]
19. Schlichting, R.D.; Schneider, F.B. Fail-stop processors: An approach to designing fault-tolerant distributed computing systems. *ACM Trans. Comput. Syst.* **1985**, *1*, 222–238. [CrossRef]
20. Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* **1978**, *21*, 558–565. [CrossRef]
21. Chandy, K.M.; Lamport, L. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.* **1985**, *3*, 63–75. [CrossRef]
22. Netzer, R.H.B.; Xu, J. Necessary and sufficient conditions for consistent global snapshots. *IEEE Trans. Parallel Distrib. Syst.* **1995**, *6*, 165–169. [CrossRef]
23. 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]
24. Andrews, G.R. Paradigms for process interaction in distributed programs. *ACM Comput. Surv.* **1991**, *23*, 49–90. [CrossRef]