electronics

MDPI

Article

Scalable Communication-Induced Checkpointing Protocol with Little Overhead for Distributed Computing Environments

Jinho Ahn

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

Abstract: The existing communication-induced checkpointing protocols may not scale well due to their slow acquisition of the most recent timestamps of the next checkpoints of other processes. Accurate situation awareness with diversified information conveyance paths is needed to reduce the number of unnecessary forced checkpoints taken as few as possible. In this paper, a scalable communication-induced checkpointing protocol is proposed to considerably cut down the possibility of performing unnecessary forced checkpointing by exploiting the beneficial features of reliable communication channels. The protocol enables the sender of an application message to swiftly attain the most recent timestamp-related information of the next checkpoint of its receiver and accelerate the spread of the information to others, with little overhead. This behavioral feature may significantly elevate the accuracy of the awareness of the situations in which forced checkpointing is actually needed for useless checkpoint-free recovery. In addition, it generates no extra control message and no message logging overhead while significantly lessening the latency of message sending. Moreover, the protocol can always be operated under the non-deterministic execution model. The evaluation results indicate that the proposed protocol outperforms the existing ones at the reduced forced checkpointing overheads from 12.5% to 84.2%, and at the reduced total execution times from 2.5% to 11.5%.

Keywords: distributed systems; fault-tolerance; checkpointing; recovery; scalability

check for updates

Citation: Ahn, J. Scalable Communication-Induced Checkpointing Protocol with Little Overhead for Distributed Computing Environments. *Electronics* **2023**, *12*, 2702. https://doi.org/10.3390/electronics12122702

Academic Editor: Padmanabhan Balasubramanian

Received: 26 April 2023
Revised: 6 June 2023
Accepted: 14 June 2023
Published: 16 June 2023

1. Introduction

Communication-induced checkpointing (hereafter, CIC) is a rollback recovery technique that requires no explicit synchronization procedure with other processes upon checkpointing and that precludes the occurrence of the domino-effect phenomenon [1]. These features can be realized by having every process piggyback on each sent message, some control information of other processes as well as itself. In addition, if needed before message delivery, additional checkpoints, called forced checkpoints, can be taken based on the local variables of the process and the information piggybacked on the messages [2]. Most CIC protocols [3–15] with these advantages have the following common behavioral features. If each process recognizes the delivery of a message may make any other local checkpoint of another process, called a basic checkpoint, become useless, the protocols make the process take a forced checkpoint.

CIC protocols are classified into two types, model-based CIC and timestamp-based CIC [1]. Although model-based protocols [3,4] may be transformed to their corresponding timestamp-based ones [5–15], the overhead of the first, in its eagerness to break suspicious message exchange patterns, can be considerably higher than that of the second in terms of the number of forced checkpoints [1]. Timestamp-based protocols have continuously advanced with their protocol-specific checkpoint timestamping schemes using Lamport's logical clock to decrease the number of forced checkpoints in the following manner. Early-stage CIC protocols [5,6] make each process take a forced checkpoint if their timestamp-increasing flow condition is not satisfied when comparing its local timestamp

with the one piggybacked on each message received. However, this may cause a large number of unnecessary forced checkpoints due to the inaccuracy in decisions about the uselessness of basic checkpoints of other processes. To cut back on the overhead as much as possible, improved protocols [7,8,11–15] have been designed to utilize more information piggybacked on each message, including the most recent checkpoint timestamp of every other process. In particular, among them, a representative protocol [7], HMNR, attempts to lower the forced checkpointing overhead by narrowing the condition inducing forced checkpoints with the help of the control information contained in each message received. A recently enhanced version of HMNR, LazyHMNR [8], is designed to balance the growing rates of checkpoint timestamps of processes by temporarily ceasing to increase the timestamps taken of outliers.

However, despite efforts to decrease the frequency of forced checkpointing, these previous protocols have a much greater number of forced checkpoints than basic checkpoints, highly increasing additional execution time and the stable storage size required [16]. This results in the negative effect of slower acquisition of the most recent timestamps of the next checkpoints of other processes when receiving each message. Accurate situation awareness with diversified information conveyance paths is needed to make the number of unnecessary forced checkpoints taken as low as possible.

Although developed to operate based on a reliable first-in, first-out (hereafter, FIFO) communication channel, most traditional CIC protocols [3–8,10–15] do not use this advantageous feature to lessen this sort of forced checkpointing overhead. A previous CIC protocol [9] attempted to decrease the number of forced checkpoints by obtaining the up-to-date local clock of the next checkpoint of every other process based on performing sender-based message logging earlier than before. However, the protocol may not only generate a few extra control messages but may also greatly lengthen the latency of message sending. In addition, it has a critical limitation that, due to its usage of message logging, the applicable process execution model can be piecewise deterministic only. Another most recent CIC protocol [10] is designed to use pessimistic message logging to cut down on the forced checkpointing frequency. The protocol can overcome the constraint of the piecewise deterministic model by recognizing which execution point is deterministic in each checkpoint interval. However, it has the same limitation as the traditional ones [3–8,11–15] mentioned above—that is, the slow gathering of the most recent local timestamp of the next checkpoint of every other process. In addition, if non-deterministic events that cannot be logged often occur in the execution of processes, the performance gain may be greatly diminished.

This paper proposes a scalable CIC protocol to considerably cut down the possibility of performing unnecessary forced checkpointing by exploiting the beneficial feature of reliable communication channels generally assumed in the research field, namely, rollback recovery with checkpointing and message logging. This protocol enables each process sending a message to attain the most recent timestamp-related information of the next checkpoint of its receiver as fast as possible and accelerate the spread of the information to others with little overhead. For this purpose, the information is piggybacked on the acknowledgment of each application message essential to ensuring communication reliability, which requires no additional control message. In addition, it allows each process to perform actual message sending operations right after receiving messages without any delay, unlike the existing protocols [9]. Moreover, the protocol is never constrained by the piecewise deterministic execution model because it does not utilize message logging.

2. Background

2.1. Assumptions

In a distributed system consisting of a set of n processes following the crash failure model, the processes exchange messages with each other through reliable FIFO communication channels [17]. Each process p takes its i-th local checkpoint ($i \geq 0$), Ck_p^i, to record its current state in stable storage for recovery. At this time, p increments its local clock, cl_p, by

one and assigns it to Ck_p^i, denoted by $\mathrm{Ck}_p^i.cl$. Whenever p sends a message m to another process q after Ck_p^i, it piggybacks cl_p on m (denoted by $m.cl$). CI_p^i stands for a checkpoint interval composed of a set of events of internal computing, message sending, and receiving performed by p from Ck_p^i to Ck_p^{i+1} with Ck_p^i and without Ck_p^{i+1}. In addition, some generally used terminologies related to consistency on checkpoints are defined as follows:

Definition 1. *A global checkpoint is a set of n local checkpoints consisting of only one of each process in the system [18,19].*

Definition 2. *An orphan message is a message which is received but never sent between a pair of local checkpoints of two different processes during recovery [20,21].*

Definition 3. *The consistency of a global checkpoint is ensured if and only if there exists no orphan message between every pair of local checkpoints belonging to the global checkpoint [22].*

The previous relation $\rightarrow^{hb}$ does not hold sufficient capability to decide whether two causally unrelated checkpoints belong to a consistent global checkpoint [10]. Figures 1 and 2 illustrate the limitation of the relation. In Figure 1, a checkpoint timestamping function using Lamport's logical clock allows the system to identify that Ck_p^i and Ck_r^{k+1}($\mathrm{Ck}_p^i \rightarrow^{hb} \mathrm{Ck}_r^{k+1}$) do not belong to the same consistent global checkpoint through two causally related messages m_1 and m_2. Here, two global checkpoints (drawn in red and blue colors), (Ck_p^i, Ck_q^j, Ck_r^{k+1}) and (Ck_p^i, Ck_q^{j+1}, Ck_r^{k+1}), are both inconsistent because m_2 and m_1 become orphan messages, in order. However, though $\mathrm{Ck}_p^i \nrightarrow^{hb} \mathrm{Ck}_r^{k+1}$ in Figure 2, both are also inconsistent through two causally unrelated messages m_1 and m_2. To overcome this insufficiency, the notion of a Z-path [2] has been devised to detect the uncaught message patterns (called NC-patterns) between a pair of local checkpoints.

Definition 4. *A Z-path from Ck_p^i to Ck_q^j (denoted by $Ck_p^i \rightarrow^{zz} Ck_q^j$) is a message path composed of a sequence of messages [m_1, m_2, ..., m_k]($k \geq 1$) if and only if every following condition is satisfied [2]:*

- *After having taken Ck_p^i, p sends a message m_1;*
- *For $\forall w(1 \leq w < k)$, both receiving m_w and sending m_{w+1} are executed by another process r in the same checkpoint interval, or there is at least one checkpoint after the first and before the second;*
- *q receives the message m_k before taking Ck_q^j.*

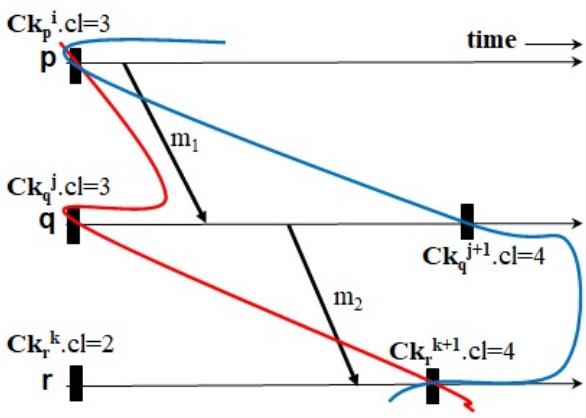

Figure 1. Example of causal path from Ck_p^i to Ck_r^{k+1}.

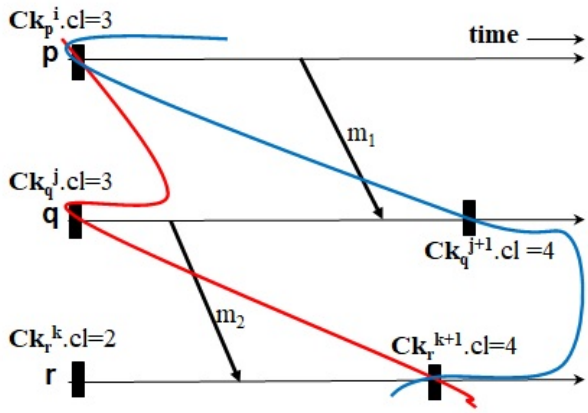

Figure 2. Example of causally unrelated path from Ck_p^i to Ck_r^{k+1}.

Definition 5. *Two consecutive messages in a Z-path form an NC-pattern if sending one of them happens before receiving the other at the same checkpoint interval [7].*

A Z-path can be classified into two kinds, a C-path and an NC-path. A C-path is a message sequence where the delivery of the message to each process except the last one always precedes the message being sent from the same process, such as [m_1, m_2] in Figure 1.

Definition 6. *A Z-path with no NC-pattern is called a C-path [7].*

An NC-path is a message sequence where the ordering rule of C-path for delivering and sending messages from each process is violated, such as [m_1, m_2, m_3] in Figure 3.

Definition 7. *A Z-path including at least one NC-pattern is called an NC-path [7].*

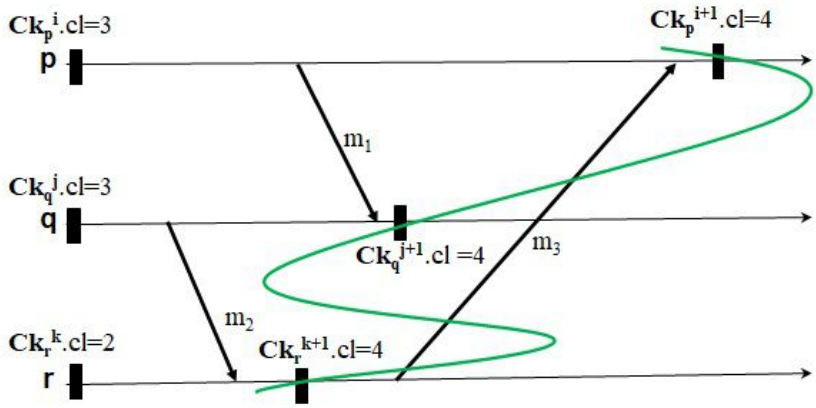

Figure 3. Example of an NC-path consisting of m_1, m_2, and m_3.

In order to reduce the rollback distance of each process as much as possible in case of process failures, the protocol periodically takes its local checkpoint on its own speed. However, even if the checkpoint timestamping function is applied in the system, this checkpointing autonomy may result in the checkpoint not belonging to any consistent global checkpoint. This useless checkpoint may result in a great rollback distance of not only the crashed process but also the other processes during recovery.

Definition 8. *A local checkpoint belonging to no consistent global checkpoint is called a useless checkpoint [7].*

A Z-path flowing from a checkpoint back to itself, called a Z-cycle, prevents the checkpoint from being part of any consistent global checkpoint.

Definition 9. *A Z-cycle is a Z-path holding $Ck_p^i \rightarrow^{zz} Ck_p^i$ [2].*

The property of Z-cycles may become a precise condition for determining whether each checkpoint is useless. For example, Figure 3 shows a Z-path consisting of an NC-pattern [m_1,m_2] from Ck_p^i to Ck_r^{k+1} and then a message m_3 after Ck_r^{k+1}, where Ck_r^{k+1} comes back to itself, called a Z-cycle, [m_3,m_1,m_2]. In this example, Ck_r^{k+1} becomes useless because no global checkpoint including Ck_r^{k+1} is consistent, such as (Ck_p^i, Ck_q^j, Ck_r^{k+1}), (Ck_p^i, Ck_q^{j+1}, Ck_r^{k+1}), and (Ck_p^{i+1}, Ck_q^{j+1}, Ck_r^{k+1}) in Figure 3.

Theorem 1. *Ck_p^i is a useless checkpoint if and only if it forms $Ck_p^i \rightarrow^{zz} Ck_p^i$ [2,3,7].*

Therefore, if it is guaranteed that every local checkpoint never becomes useless, the Z-cycle-free system can be recovered with the most recent consistent global state after process failure, decreasing the amount of computation nullified due to the crashes to the maximum extent.

Corollary 1. *A system is Z-cycle-free if no useless checkpoint exists.*

2.2. Related Work

Most existing CIC protocols [3–10,14,15] have been designed to ensure that no basic checkpoint becomes useless by satisfying Theorem 2 at all times.

Theorem 2. *For any pair of checkpoints Ck_p^i and Ck_q^j, $Ck_p^i \rightarrow^{zz} Ck_q^j$ with ($Ck_p^i.cl < Ck_q^j.cl$) includes no Z-cycle [1,2].*

Theorem 2 provides the criteria for designing a CIC protocol that ensures useless checkpoints involved in a Z-cycle are not created by keeping the checkpoint timestamp flow increasing along any Z-path.

Let us review how the previous works advanced CIC protocols with some examples in Figures 4 and 5. Early-stage CIC protocols [5,6] let each process q take a forced checkpoint if the following equation is met in order to conform to Theorem 2 (such as in Figure 4):

$$m_1.cl > m_2.cl \tag{1}$$

Equation (1) can be implemented with two vector variables, $sent\_to_q$ and min_q. The first is used to detect the creation of NC-patterns in every other process r. In other words, $sent\_to_q[r]$ is a Boolean variable that is set to true if q sends a message m_2 to r for the first time since its latest checkpoint. $min_q[r]$ maintains the checkpoint timestamp of m_2, $m_2.cl$. When q receives m_1 from another p, the following deduced safety predicate, $C_{early}^{m_1}$, is performed to evaluate whether q should take a forced checkpoint:

$$C_{early}^{m_1} \equiv sent\_to_q[r] \wedge (m_1.cl > min_q[r]).$$

However, the aggressive predicate has no functionality and cannot obtain the most recent checkpoint timestamp of every other process. This insufficiency of information acquisition may incur high failure-free costs by taking unnecessary forced checkpoints, such as in Figure 4. In this example, when q receives m_1 from p, it can recognize an NC-pattern connecting m_1 with m_2 because $sent\_to_q[r]$ is true. In addition, as $m_1.cl$ (= 3) > $min_q[r]$ (= 2), the value of $C_{early}^{m_1}$ is true, and so q takes a forced checkpoint Ck_q^{j+1}. However, in this case, it is not necessary that q performs the forced checkpointing because $\mathrm{Ck}_p^i.cl < \mathrm{Ck}_r^{k+1}.cl$.

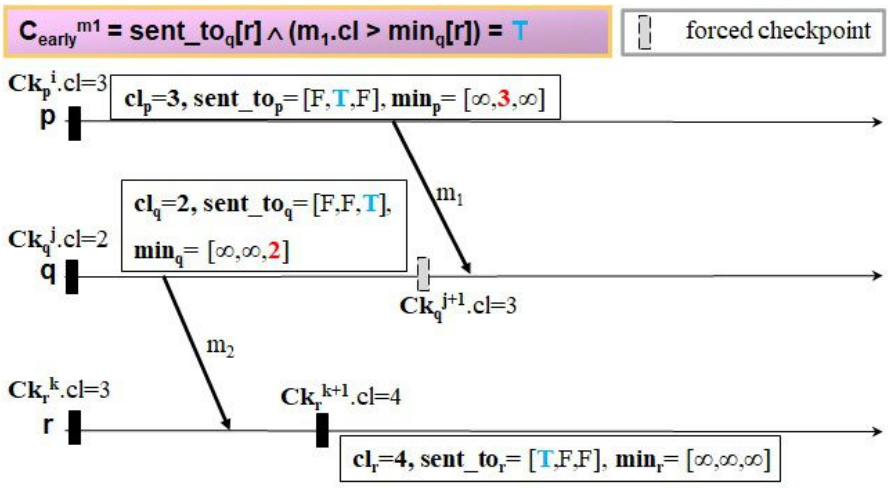

Figure 4. Example of removing the potential of Z-cycle occurrences in advance in early-stage protocols.

A set of HMNR protocols [7,8,14,15] is designed to reduce the number of unnecessary forced checkpoints by narrowing down Equation (1) to Equation (2), as follows, where $cl_q(r)$ is the value of the most recent checkpoint timestamp of r that q knows:

$$(m_1.cl > m_2.cl) \wedge ((m_1.cl > cl_q(r)) \vee (cl_q(r) \geq Ck_r^{k+1}.cl)) \tag{2}$$

Equation (2) can also be split into two Equations (3) and (4):

$$(m_1.cl > m_2.cl) \wedge (m_1.cl > cl_q(r)) \tag{3}$$

$$(m_1.cl > m_2.cl) \wedge (cl_q(r) \geq Ck_r^{k+1}.cl) \tag{4}$$

To implement Equation (3), aside from $sent\_to_p$ and cl_p, a process p keeps a lightweight vector $greater_p$ to indicate whether its current logical clock cl_p is greater than the latest checkpoint timestamp of every other p estimate. In other words, $greater_p[r]$ is a Boolean variable that is set to false when receiving m_α from r because $m_\alpha.cl \geq cl_p$ in Figure 5. The vector variable is piggybacked on each sent message m, denoted by $m.greater$. As q receives m_1 from p, the following safety predicate, $C_1^{m_1}$, is conducted for q to make a decision about whether a forced checkpointing action should be performed:

$$C_1^{m_1} \equiv sent\_to_q[r] \wedge m_1.greater[r] \wedge (m_1.cl > cl_q)$$

Let us clarify how the HMNR protocols require fewer forced checkpoints compared with the early-stage ones with the example in Figure 5. In this example, the current checkpoint timestamp of r flows to p through m_α between m_2 and m_1. As $m_\alpha.cl$ and cl_p have the same value(=3), $greater_p[r]$ becomes false and is relayed to q through m_1 as $m_1.greater[r]$. At this point, q does not take any forced checkpoint because the value of $C_1^{m_1}$ is false, i.e., $\mathrm{Ck}_p^i.cl < \mathrm{Ck}_r^{k+1}.cl$. After p and q eventually take their basic checkpoints, Ck_p^{i+1} and Ck_q^{j+1}, respectively, at their preferred execution positions, a globally consistent checkpoint (Ck_p^{i+1}, Ck_q^{j+1}, Ck_r^{k+1}) is formed. However, if there is no application message such as m_α between m_1 and m_2 in Figure 5, the protocols should still perform forced checkpointing like in Figure 4. This limitation may considerably and unnecessarily increase additional application execution time and stable storage size needed for recovery.

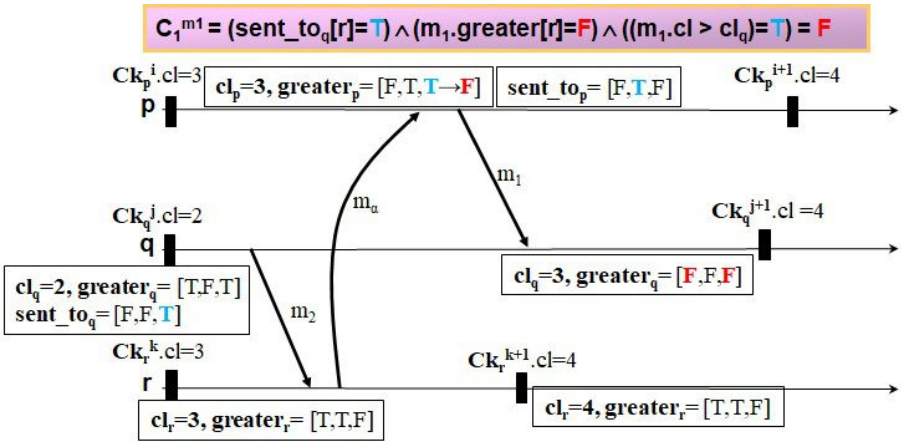

Figure 5. Example of decreasing forced checkpointing frequency by piggybacking in a family of HMNR protocols.

Equation (4) can be implemented with two vector variables, $ckpt_q$ and $taken_q$. $ckpt_q$ is a table used to record the number of checkpoints of each process taken since its beginning that q knows. $taken_q$ is a Boolean vector indicating whether there is a C-path to the next checkpoint of q from the latest checkpoint of every other process that q presumes. When q takes a local checkpoint, all elements in $taken_q$, except one for itself, which is always false, are reset to true. Whenever an application message is sent, the two variables are included in the message. If p sends m_1 to another process q such as in Figure 6, q carries out the following safety predicate, $C_2^{m_1}$, to determine whether m_1 may potentially cause a Z-cycle to be formed:

$$C_2^{m_1} \equiv (ckpt_q[q] = m_1.ckpt[q]) \wedge m_1.taken[q].$$

Figure 6 illustrates an instance of taking a forced checkpoint with $C_2^{m_1}$ in the HMNR protocols. When receiving m_2, r updates $ckpt_r[q]$ and $taken_r[q]$ to j and false, respectively. If r takes a basic checkpoint Ck_r^{k+1}, $taken_r[q]$ is reset to true. Receiving m_β, p updates $ckpt_p[q]$ and $ckpt_p[r]$ to j and $k+1$ and $taken_p[r]$ to false. When q receives m_1, it decides to perform forced checkpointing as the value of $C_2^{m_1}$ is true.

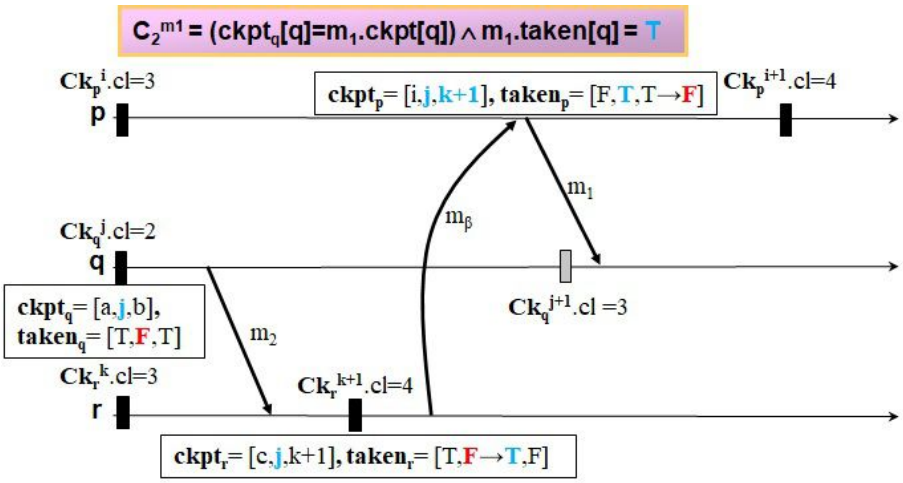

Figure 6. Example of performing forced checkpointing by satisfying Equation (4) in a family of HMNR protocols (a, $c < i$ and $b < k$).

To reduce the forced checkpointing overhead, a recent CIC protocol [9] with sender-based message logging, EsbmlCIC, attempts to obtain the up-to-date local clock of the next checkpoint of every other process by using its control message exchange procedure to notify the sender of each application message of its received sequence number. However,

due to the behavioral feature of sender-based message logging, the procedure always requires one extra control message per application and may also incur a not little delay on completing message send operations requested right after the receipt of the application. Moreover, it should be subject to the piecewise deterministic model because it employs message logging.

Another recent CIC protocol [10], S-CIC, attempts to bring the forced checkpointing frequency down as much as possible by using pessimistic message logging without being restricted by the piecewise deterministic model. This can be achieved by identifying which part of each checkpoint interval is deterministic. Even if receiving a message m induces a forced checkpoint, as either the value of C_1^m or C_2^m is true, the protocol does not perform the checkpointing action at the execution point of the sender right before transmitting m is recoverable. However, if there exists at least one non-deterministic event that cannot be replayed in the interval right before sending m as well as in every interval which the first depends on, the protocol should take the same forced checkpoint as the previous ones. In addition, this protocol does not have the functionality to acquire the latest timestamps of the next checkpoints of other processes earlier than the latter.

Unlike all the CIC protocols stated above, three CIC protocols, Adaptive [11], FINE [12], and LazyFINE [13], cannot realize useless checkpoint-free recovery. They sacrifice this feature in exchange for a lower forced checkpointing frequency by weakening the condition identifying the Z-cycle pattern, as shown in Figure 3.

3. The Scalable CIC Protocol

The proposed CIC protocol, LightweightCIC, is designed to substantially decrease the possibility of making wrong decisions when performing forced checkpointing, as follows:

- The protocol enables each process sending a message to attain the most recent timestamp information of the next checkpoint of its receiver as fast as possible and accelerate the spread of the information to others in a lightweight manner.
- It incurs neither additional control message nor delay in the completion of application message send operations executed right after each message is received.
- It can always be operated under the non-deterministic execution model.

The protocol can meet the third requirement because it is a checkpoint-only protocol using no message logging technique unlike our previous one [9]. Then, to combine the first and second requirements together, it exploits the beneficial feature of reliable communication channels, as follows. The protocol has each process r receiving a message m_2 from another process q piggyback its current timestamp ($cl_r(r)$) upon acknowledgment of m_2, denoted by ack_2. Then, it propagates the information, $ack_2.cl$, not only to its sender q but also to others in an effective manner. With this immediate update of the information, the protocol identifies whether q should take a forced checkpoint upon receipt of a message m_2 based on Equations (4) and (5), unlike the HMNR protocols [7,8,14,15]:

$$(m_1.cl > ack_2.cl) \wedge (m_1.cl > cl_q(r)) \tag{5}$$

Compared with Equation (3), Equation (5) can greatly enhance the up-to-dateness of the two important variables of each process q, cl_q and $greater_q$, and speed up their dissemination to other processes. This improvement may greatly boost the accuracy of both $m_1.greater[r]$ and $(m_1.cl > cl_q)$ in $C_1^{m_1}$. With this feature, the protocol can greatly reduce the possibility of taking unnecessary forced checkpoints compared to the existing protocols [3–15]. We proved the superiority of Equation (5) over Equation (3) in Lemma 1 in terms of the number of forced checkpoints taken.

Lemma 1. *Equation (5) causes fewer or the same forced checkpoints as Equation (3) to be taken.*

Proof. We prove the correctness of the lemma by contradiction. It is assumed that Equation (5) may lead to more forced checkpoints being taken than Equation (3), meaning the first

equation cannot detect the situations where forced checkpointing need not be performed more accurately than the latter, as Theorem 2 is satisfied. Suppose two consecutive messages m_1 and m_2 form an NC-pattern where sending m_2 to r after Ck_r^k before Ck_r^{k+1} precedes receiving m_1 from p after Ck_p^i before Ck_p^{i+1} in the same checkpoint interval CI_q^j. There are two cases that can happen, as follows:

Case 1: $m_1.cl \leq m_2.cl$.

In this case, as $\mathrm{Ck}_p^i.cl \leq m_2.cl < \mathrm{Ck}_r^{k+1}.cl$, both equations enable q to skip performing a forced checkpointing action when q receives m_1.

Case 2: $m_1.cl > m_2.cl$.

In this case, we should consider two subcases, as follows:

Case 2.1: $m_2.cl \geq cl_r(r)$ right after m_2.

In this case, as $\mathrm{Ck}_p^i.cl \not< \mathrm{Ck}_r^{k+1}.cl$, both equations cause a forced checkpoint to be taken when q receives m_1.

Case 2.2: $m_2.cl < cl_r(r)$ right after m_2.

In this case, we should consider the following two subcases:

Case 2.2.1: $m_1.cl > cl_r(r)$ right after m_2.

In this case, as $\mathrm{Ck}_p^i.cl \not< \mathrm{Ck}_r^{k+1}.cl$, both equations force q to decide to perform a forced checkpointing action when q receives m_1.

Case 2.2.2: $m_1.cl \leq cl_r(r)$ right after m_2.

In this case, $\mathrm{Ck}_p^i.cl \leq m_1.cl < \mathrm{Ck}_r^{k+1}.cl$. Equation (5) can detect this situation because $ack_2.cl$ is equal to $cl_r(r)$ right after m_2 and results in no forced checkpoint being taken when q receives m_1. However, Equation (3) cannot identify this situation without the help of a C-path starting with m_2 and bringing $cl_r(r)$ to q before q actually delivers m_1. Due to this limitation, Equation (3) may exclude this situational possibility to skip taking a forced checkpoint.

Therefore, Equation (5) leads to fewer or the same number of forced checkpoints to be taken as in Equation (3). This contradicts the hypothesis. □

Figure 7 shows the formal description of five modules of the proposed protocol for each process p. The first module, INITIALIZE(), initializes all the variables that p should maintain and takes its initial checkpoint. The second module, LOCAL-CHECKPOINTING(), increments both its checkpoint timestamp and the number of checkpoints taken by p from the beginning of its execution by one. Then, it resets several other variables used for deciding whether the value of $C_1^{m_1}$ is true and records its current state with the first two variables in the stable storage. The third module, MSG-SEND(), indicates that p sends a message m to the receiver of m after its most recent checkpoint. Then, p sends the message with the variables for making a decision about whether the receiver is forced to take a checkpoint before delivering m. The fourth module, MSG-RECV(), first checks whether the received message may make useless checkpoints. If so, p performs a forced checkpointing by calling LOCAL-CHECKPOINTING(). Then, p sends an acknowledgment with its checkpoint timestamp and $greater_p$. Next, it updates its local variables related to the decision-making condition according to the values of the piggybacked variables and delivers the contents of the message to its target application. The last module, ACK-RECV(), updates its checkpoint timestamp and $greater_p$ according to the value of the checkpoint timestamp piggybacked on the acknowledgment.

```
Module INITIALIZE() AT p
    cl_p ← 0;
    greater_p[p] ← F;
    taken_p[p] ← F;
    ∀j(1≤j≤n): ckpt_p[j] ← 0;
    call LOCAL-CHECKPOINTING();

Module LOCAL-CHECKPOINTING() AT p
    increment cl_p by one;
    increment ckpt_p[p] by one;
    ∀j(1≤j≤n): sent_to_p[j] ← F;
    ∀j(1≤j≤n, j ≠ p): greater_p[j] ← T;  taken_p[j] ← T;
    save its local state with (cl_p, ckpt_p[p]) on the stable storage;

Module MSG-SEND(data, q) AT p
    if(sent_to_p[q] = F) then sent_to_p[q] ← T;
    send m(cl_p, greater_p, ckpt_p, taken_p, data) to q;

Module MSG-RECV(m(cl, greater, ckpt, taken, data)) AT p from q
// Check whether a forced checkpoint should be taken before m. //
    if(((∃j(1≤j≤n):sent_to_p[j] ∧ m.greater[j]) ∧ (m.cl > cl_p)) ∨
      ((ckpt_p[p] = m.ckpt[p]) ∧ m.taken[p])) then
      call LOCAL-CHECKPOINTING();
// Update its local clock and Z-cycle detection variables. //
    if(m.cl > cl_p) then
      send ack(cl_p, ⊥) to q;
      cl_p ← m.cl;
      ∀j(1≤j≤n, j ≠ p): greater_p[j] ← m.greater[j];
    else if(m.cl = cl_p) then
      send ack(cl_p, greater_p) to q;
      ∀j(1≤j≤n, j ≠ p): greater_p[j] ← greater_p[j] ∧ m.greater[j];
    else
      send ack(cl_p, greater_p) to q;
      greater_p[q] ← F;
    ∀j(1≤j≤n, j ≠ p):
      if(m.ckpt[j] > ckpt_p[j]) then
        ckpt_p[j] ← m.ckpt[j];  taken_p[j] ← m.taken[j];
      else if(m.ckpt[j] = ckpt_p[j]) then
        taken_p[j] ← taken_p[j] ∨ m.taken[j];
    deliver m.data to its corresponding application;

Module ACK-RECV(ack(cl, greater)) AT p from q
    if(ack.cl > cl_p) then
      cl_p ← ack.cl;
      ∀j(1≤j≤n, j ≠ p): greater_p[j] ← ack.greater[j];
    else if(ack.cl = cl_p) then
      ∀j(1≤j≤n, j ≠ p): greater_p[j] ← greater_p[j] ∧ ack.greater[j];
    else // the case that cl_q has been already updated with the value of cl_p as cl_q < cl_p. //
      greater_p[q] ← F;
```

Figure 7. Modules of LightweightCIC for process p.

Let us examine how our protocol can fulfill both requirements with the beneficial feature unlike a representative useless checkpoint-free CIC protocol, HMNR, using several examples of checkpointing and communication patterns. Figures 8–13 illustrate four examples to clarify how the two protocols trigger events of taking forced checkpoints with three messages, m_1, m_2, and m_3. The first example, shown in Figures 8 and 9, shows the

case where $\mathrm{Ck}_q^j.cl < \mathrm{Ck}_p^i.cl < \mathrm{Ck}_r^k.cl$. In Figure 8, HMNR takes two unnecessary forced checkpoints, as follows. When receiving m_1, q decides to record a forced checkpoint state Ck_q^{j+1} before m_1 because it has a mistaken knowledge that $\mathrm{Ck}_p^i.cl$ is equal to that of the next checkpoint of r (i.e., the value of $C_1^{m_1}$ is true). Similarly, as p receives m_3 from r, it performs forced checkpointing because it knows $\mathrm{Ck}_r^k.cl$ is equal to that of the next checkpoint of q after having received m_1 (i.e., the value of $C_1^{m_3}$ is true). However, in Figure 9, LightweightCIC triggers no events of forced checkpointing for the reasons mentioned below. On the receipt of m_2, r knows its current logical clock value is greater than cl_q piggybacked on m_2 (on calling module MSG-RECV()). Then, it conveys to q the acknowledgment of m_2 with cl_r and $greater_r$ and indicates that the updated checkpoint timestamp of q is equal to cl_r($greater_r[q]$←F). Afterwards, q changes cl_q and $greater_q[r]$ to the current timestamp of r piggybacked on the acknowledgment and $ack_2.greater[r]$ (on calling module ACK-RECV()). When receiving m_1, q can accurately recognize it is not necessary for q to take a forced checkpoint as $\mathrm{Ck}_p^i.cl < \mathrm{Ck}_r^{k+1}.cl$ (i.e., the value of $C_1^{m_1}$ is false) (on calling module MSG-RECV()). Then, as q knows cl_p is less than cl_q, it immediately transmits the acknowledgment of m_1 with cl_q and $greater_q$ to p and sets $greater_q[p]$ to F. Upon receiving this message, p updates cl_p and every $greater_p[i](i \neq p)$ with $ack_1.cl$ and $ack_1.greater[i]$, respectively, (on calling module ACK-RECV()), like in Figure 9. When receiving m_3, LightweightCIC restrains p from performing forced checkpointing because it knows $\mathrm{Ck}_r^k.cl < \mathrm{Ck}_q^{j+1}.cl$ (i.e., the value of $C_1^{m_3}$ is false) (on calling module MSG-RECV()). Upon the receipt of ack_3, r changes $greater_r[p]$ to F. With this reduction in overhead, the proposed protocol requires no extra control message to quickly update the recovery information and has no delay on actually sending messages right after each message is received by their corresponding targets.

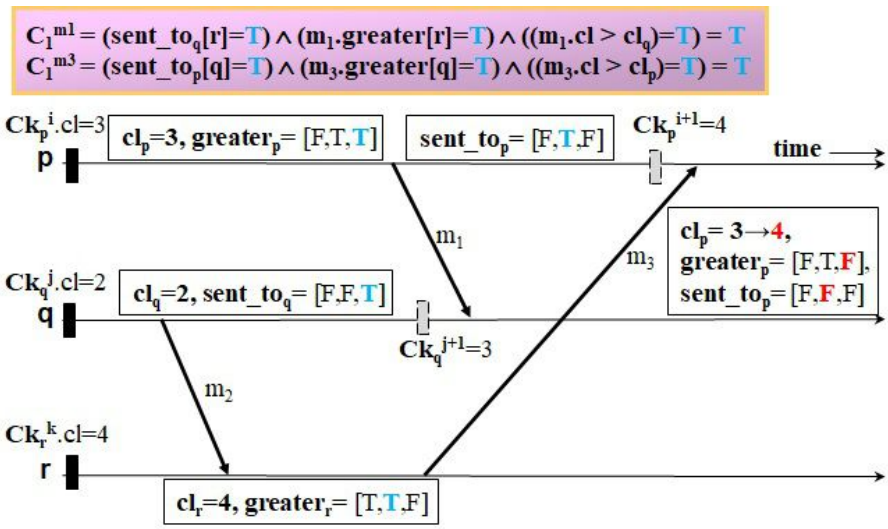

Figure 8. HMNR takes two forced checkpoints in the first example.

The second example, shown in Figures 10 and 11, shows the case that $\mathrm{Ck}_q^j.cl < \mathrm{Ck}_p^i.cl = \mathrm{Ck}_r^k.cl$. In Figure 10, upon the receipt of m_1, HMNR takes one forced checkpoint, Ck_q^{j+1}, before delivering m_1 because q wrongly believes $\mathrm{Ck}_p^i.cl = \mathrm{Ck}_r^{k+1}.cl$ after m_2. On the other hand, it lets p perform no forced checkpointing when receiving m_3, as it can recognize that $\mathrm{Ck}_r^k.cl$ is less than the timestamp of the next checkpoint of q after m_1. However, in Figure 11, LightweightCIC does not only have p, but also q does not perform forced checkpointing when receiving m_1 because q can see $\mathrm{Ck}_p^i.cl < \mathrm{Ck}_r^{k+1}.cl$ after m_2 according to the update of cl_q upon receiving ack_2.

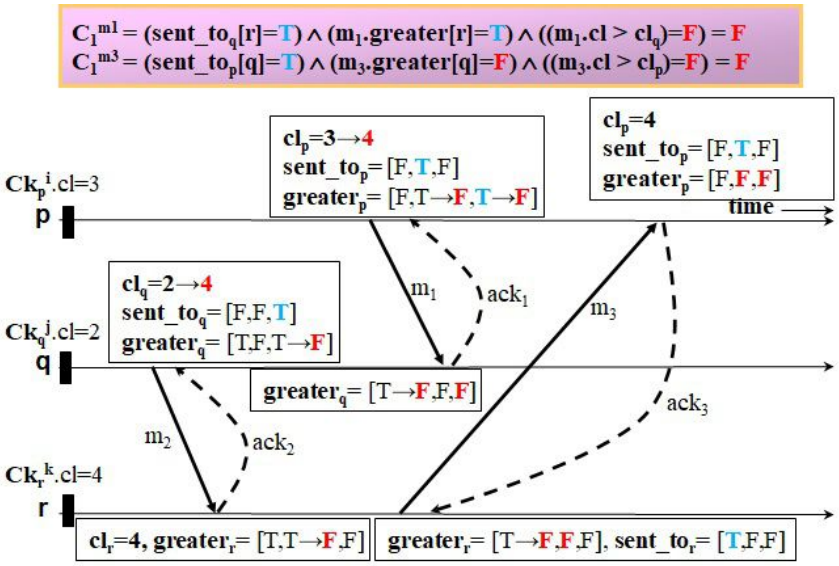

Figure 9. LightweightCIC takes no forced checkpoint in the first example.

The third example, shown in Figures 12 and 13, illustrates the case that $\mathrm{Ck}_p^i.cl = \mathrm{Ck}_q^j.cl < \mathrm{Ck}_r^k.cl$. When q receives m_1 in Figure 12, HMNR does not need to record a forced checkpoint, as it can detect $\mathrm{Ck}_p^i.cl < \mathrm{Ck}_r^{k+1}.cl$ after m_2. However, before delivering m_3, HMNR forces p to perform forced checkpointing because p cannot ensure the timestamp of the next checkpoint of q after m_1 is greater than $\mathrm{Ck}_r^k.cl$. In contrast, after q has received m_1 in Figure 13, LightweightCIC has q piggyback cl_q and $greater_q$ on ack_1 and has p update cl_p along with it. Thanks to this procedure, upon the receipt of m_3, LightweightCIC prohibits p from performing forced checkpointing, as it can sense that $\mathrm{Ck}_r^k.cl < \mathrm{Ck}_q^{j+1}.cl$ by checking $C_1^{m_3}$.

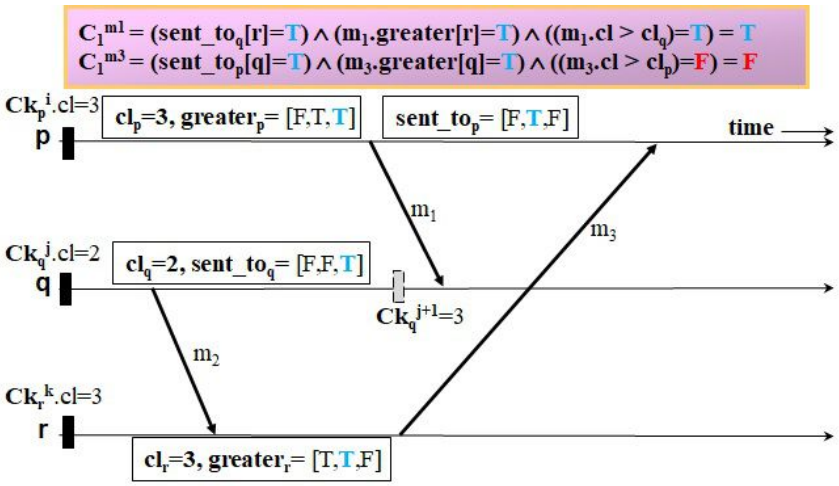

Figure 10. HMNR takes one forced checkpoint in the second example.

The fourth example, shown in Figures 14 and 15, depicts the case of a different checkpointing and communication pattern in which a C-path, $m_1 \rightarrow^{hb} m_2 \rightarrow^{hb} m_3$, forms an NC-path, $m_3 \rightarrow^{zz} m_1$. Receiving m_3 in Figure 14, HMNR has p generate a forced checkpoint Ck_p^{i+1} before m_3 because p recognizes $\mathrm{Ck}_r^k.cl \not< \mathrm{Ck}_q^{j+1}.cl$. On the other hand, when obtaining ack_2 in Figure 15, LightweightCIC enables q to change its current checkpoint timestamp and $greater_q$ to the most recent ones of r with the piggybacked information. Thanks to this early update, by checking the information piggybacked on m_3, p can see $\mathrm{Ck}_r^k.cl < \mathrm{Ck}_q^{j+1}.cl$,

as r already knew cl_q after receiving ack_2 is equal to cl_r before sending $m_3(greater_r[q] = \text{F})$; p thus decides not to take a forced checkpoint.

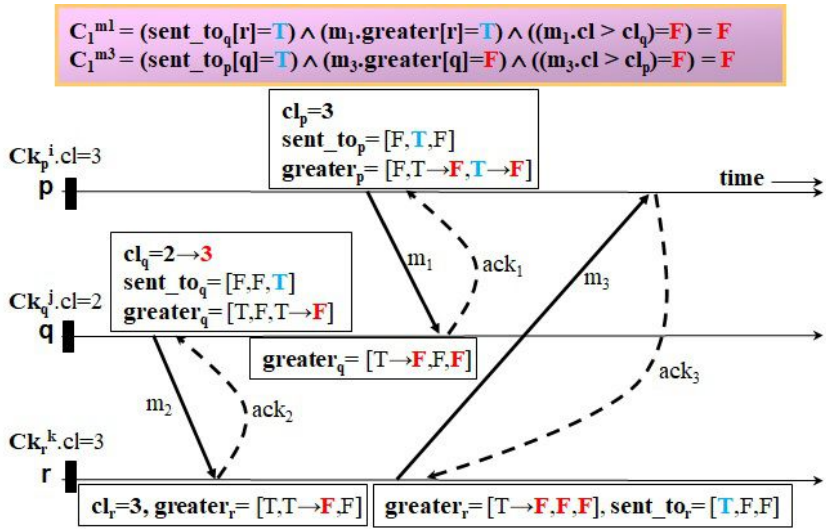

Figure 11. LightweightCIC takes no forced checkpoint in the second example.

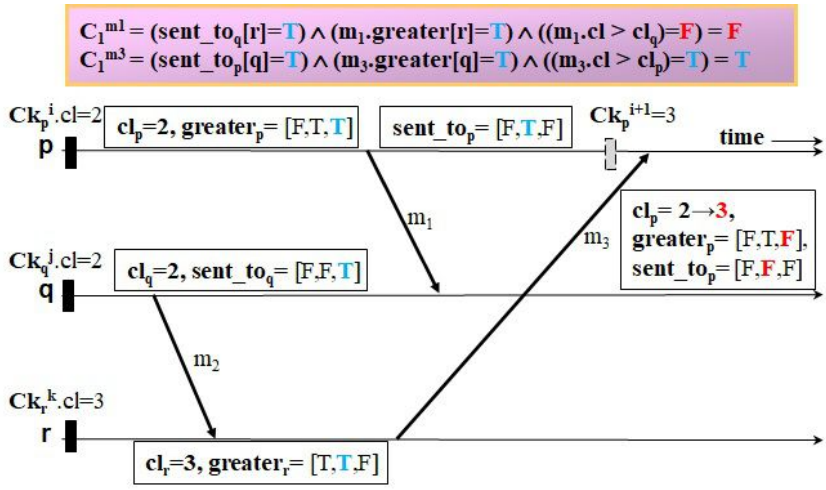

Figure 12. HMNR takes one forced checkpoint in the third example.

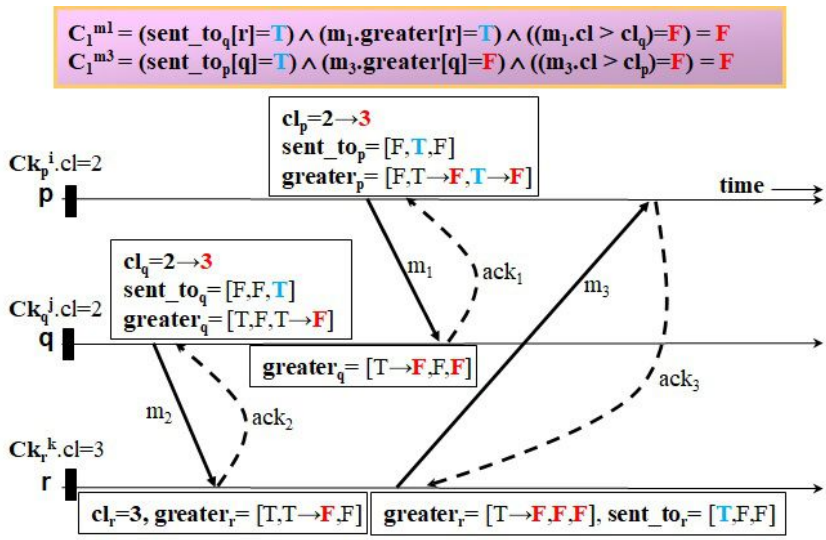

Figure 13. LightweightCIC takes no forced checkpoint in the third example.

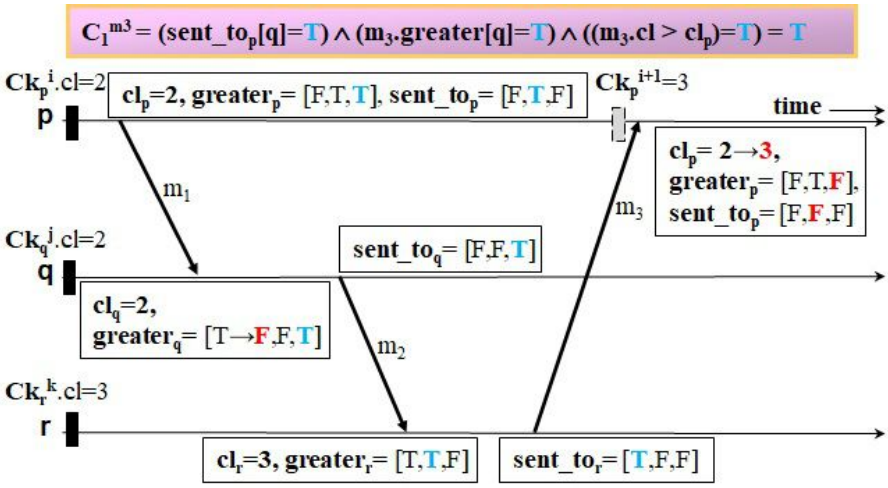

Figure 14. HMNR takes one forced checkpoint in the fourth example.

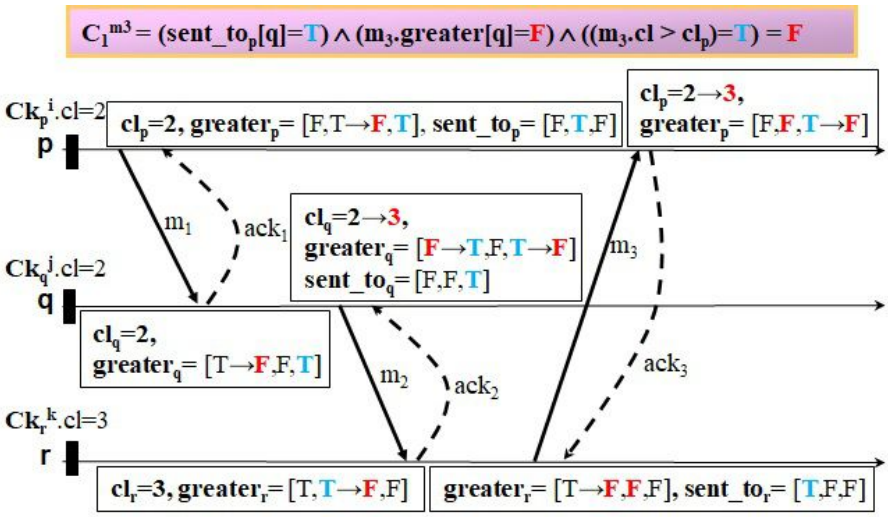

Figure 15. LightweightCIC takes no forced checkpoint in the fourth example.

Theorem 3. *Our protocol always satisfies the property that no checkpoint becomes useless.*

Proof. We prove the correctness of the theorem by contradiction. It is assume that some checkpoints become useless even after the protocol has been executed. When each process receives a message forming a Z-path, the protocol decides whether a forced checkpoint should be taken before delivering the message based on Equations (4) and (5). It was proven in HMNR [7] that the property is always satisfied in all cases where Equation (4) instructs each process not to take a forced checkpoint. Therefore, we only need to check all cases where each process decides not to take a forced checkpoint based on Equation (5). There are two cases to consider:

Case 1: the Z-path is a C-path.

In this case, as the protocol uses a checkpoint timestamping function based on Lamport's clock, the timestamp flow always goes up along any C-path. This behavioral feature makes Theorem 2 satisfied, meaning every C-path includes no Z-cycle at all times.

Case 2: the Z-path is an NC-path.

In this case, every NC-pattern in the NC-path should be checked in all cases presented in the proof of Lemma 1. Among them, we only need to examine two cases, Case 1 and Case 2.2.2, where the protocol allows each process to not take a forced checkpoint based on Equation (5). Suppose two consecutive messages m_1 and m_2 form an NC-pattern, where sending m_2 to r after Ck_r^k before Ck_r^{k+1} precedes receiving m_1 from p after Ck_p^i before Ck_p^{i+1}

in the same checkpoint interval CI_q^j.
Case 2.1: $m_1.cl \leq m_2.cl$.

In this case, as $\mathrm{Ck}_p^i.cl \leq m_2.cl < \mathrm{Ck}_r^{k+1}.cl$, delivering m_1 does not make any checkpoint useless, even if a forced checkpointing action is not performed.
Case 2.2: $m_1.cl \leq cl_r(r)$ right after m_2.

In this case, as $\mathrm{Ck}_p^i.cl \leq m_1.cl < \mathrm{Ck}_r^{k+1}.cl$, Theorem 2 is always satisfied, without taking any forced checkpoint.

Therefore, our protocol always prevents any checkpoint from being useless. This contradicts the hypothesis. □

Theorem 4. *Our protocol results in fewer or the same number of forced checkpoints being taken as HMNR.*

Proof. Our protocol instructs each process receiving a message to perform a forced checkpointing action based on Equations (4) and (5), while for HMNR, Equations (3) and (4) are used. As Equation (4) is common to both our protocol and HMNR, Equations (3) and (5) need only be compared in terms of the number of forced checkpoints taken. Based on Lemma 1, Equation (5) leads to fewer or the same number of forced checkpoints being taken as Equation (3). Therefore, the protocol has each process generate fewer or the same number of forced checkpoints as HMNR. □

4. Simulations

4.1. Experimental Environment

In this section, the effectiveness of the proposed protocol, LightweightCIC, is evaluated in comparison to the three most recent CIC protocols by using a well-known simulation language [23]. The first is the most recent protocol, LazyHMNR [8], overcoming the limitation of HMNR [7] where, if a few processes take their local checkpoints more frequently than the others, the checkpoint timestamps of the former may increase much faster than those of the latter. In this case, if the latter processes receive messages from the former, they may take quite a few of the unnecessary forced checkpoints due to the difference in their timestamps. In order to alleviate the negative effects on the high overhead caused by forced checkpointing, LazyHMNR balances the growing rates of checkpoint timestamps of processes by temporarily ceasing the increase in the timestamps of outliers. The other two are our recently presented CIC ones, S-CIC [10] and EsbmlCIC [9], mentioned in Section 2.2.

The protocols are compared using two primary performance indices, $FC_{overhead}$ and $Execution_{time}$. First, $FC_{overhead}$ means the total number of additional checkpoints needed to ensure that no basic checkpoint become useless. Second, $Execution_{time}$ represents the total execution time of each distributed application, assessed in minutes. The numerical values presented in this experiment are the averages of the results obtained through multiple runs.

For simulation, a system has n hosts which send or receive messages to and from each other. Its communication network is operated with a link capacity of 100 Mbps and a propagation delay of 1 ms. Every process in the system begins its execution on an individual host and ends together. The size of each application message destined to another process is in range of 1 KB to 1 MB. The inter-arrival time of sent messages to each process follows an exponential distribution with the average amount of time equal to three seconds. The basic checkpointing interval of each process has an exponential distribution, with the average amount of time equal to five minutes.

4.2. Experimental Outcome

As stated in Section 2.2, EsbmlCIC [9] should be operated under the piecewise deterministic execution model only. Due to this limitation, EsbmlCIC is eliminated from the first experiment, shown in Figures 16 and 17, performed in the environment where unloggable non-deterministic events are generated together with loggable non-deterministic or deterministic ones. The second experiment, indicated in Figures 18 and 19, is conducted

in the environment where each process executes message send and receive events, using deterministic processes only. Figures 16 and 17 present the $FC_{overhead}$ and $Execution_{time}$ of the three protocols, LazyHMNR, S-CIC, and LightweightCIC, in executions where the percentage of the unloggable non-deterministic events per process (UND) is 50% and the number of processes is in the range of 12 to 24. In Figure 16, the $FC_{overhead}$ is commonly proportional to the number of processes. In particular, LightweightCIC results in the lowest $FC_{overhead}$ of the three protocols, and their relative differences become greatly higher according to an increase in the number of processes. The reduction rates of $FC_{overhead}$ of LightweightCIC over LazyHMNR and S-CIC are in the range of 75.0% to 84.2% and 50.0% to 56.0%, respectively. These outcomes tell us that although S-CIC attempts to lower the number of forced checkpoints of LazyHMNR, it is much slower than LightweightCIC in acquiring the most recent timestamp-related information of the next checkpoint of each process when receiving messages. In Figure 17, we can identify that the $Execution_{time}$ of each protocol increases as the number of processes grows. Moreover, LazyHMNR has the largest $Execution_{time}$ because its excessive forced checkpointing overhead results in high application execution time. In addition, S-CIC reduces the $Execution_{time}$ of LazyHMNR by lowering the overhead with the help of pessimistic message logging and determinism detection mechanisms. However, it performs worse than LightweightCIC due to its message logging overhead and the slowness of obtaining the most recent information for decision making. The reduction rates of the $Execution_{time}$ of LightweightCIC over its two counterparts are in the range of 8.1% to 11.5% and 4.67% to 4.92%, in order.

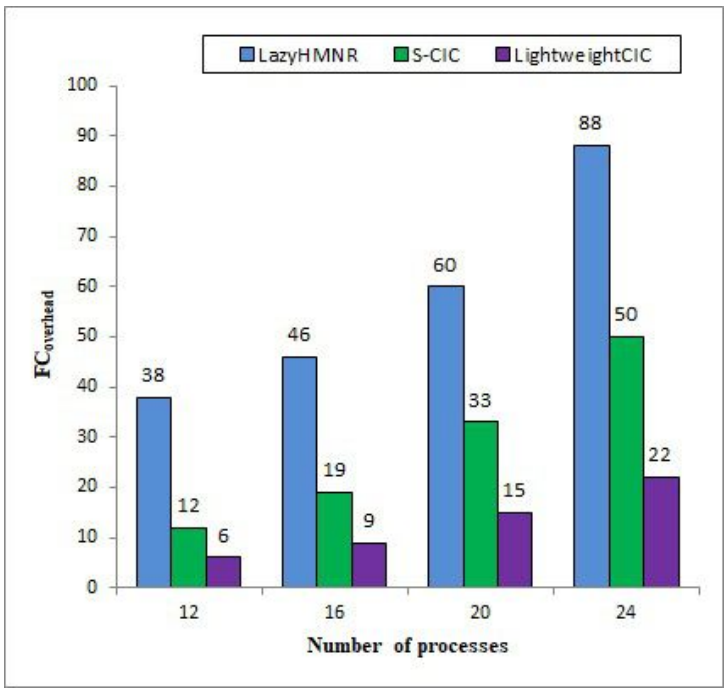

Figure 16. $FC_{overhead}$ of the three protocols in executions (UND = 50%).

Figures 18 and 19 depict the $FC_{overhead}$ and $Execution_{time}$ of the four protocols, LazyHMNR, S-CIC, EsbmlCIC, and LightweightCIC, in executions where UND is 0% and is in the same range of the number of processes as in Figures 16 and 17. In Figure 18, LazyHMNR is greatly inferior to the others in terms of $FC_{overhead}$ for the same reason as mentioned in Figure 16. On the other hand, as all the executions include loggable events only, S-CIC can considerably decrease the overhead of LazyHMNR by extending the recoverability of each process to the maximum. On the other hand, EsbmlCIC can also reduce the overhead by collecting the up-to-date local clock of the next checkpoint of every other process based on sender-based message logging earlier than the first two methods. However, LightweightCIC performs better than EsbmlCIC by intensifying the up-to-dateness of

both cl_p and $greater_p$ of each process p and speeding up their dissemination to others. The reduction rates of $FC_{overhead}$ of LightweightCIC over LazyHMNR, S-CIC, and EsbmlCIC are in the range of 75.9% to 78.6%, 25.0% to 41.2%, and 12.5% to 17.6%, respectively. Figure 19 shows the same cause and effect relationship as stated in Figure 17, i.e., LightweightCIC incurs the lowest $Execution_{time}$ of all the protocols. In this figure, EsbmlCIC is the second best in terms of $Execution_{time}$, but lags behind LightweightCIC because it not only generates more than a few extra control messages but also greatly lengthens the latency of message sending due to its behavioral feature. The reduction rates of $Execution_{time}$ of LightweightCIC over its three counterparts are in the range of 7.8% to 11.4%, 4.2% to 5.9%, and 2.5% to 3.2%, in order.

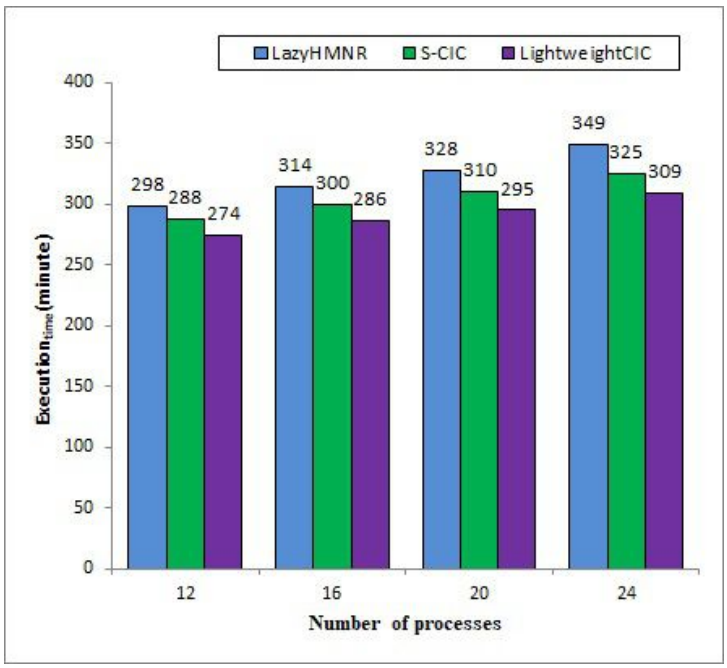

Figure 17. $Execution_{time}$ of the three protocols in executions (UND = 50%).

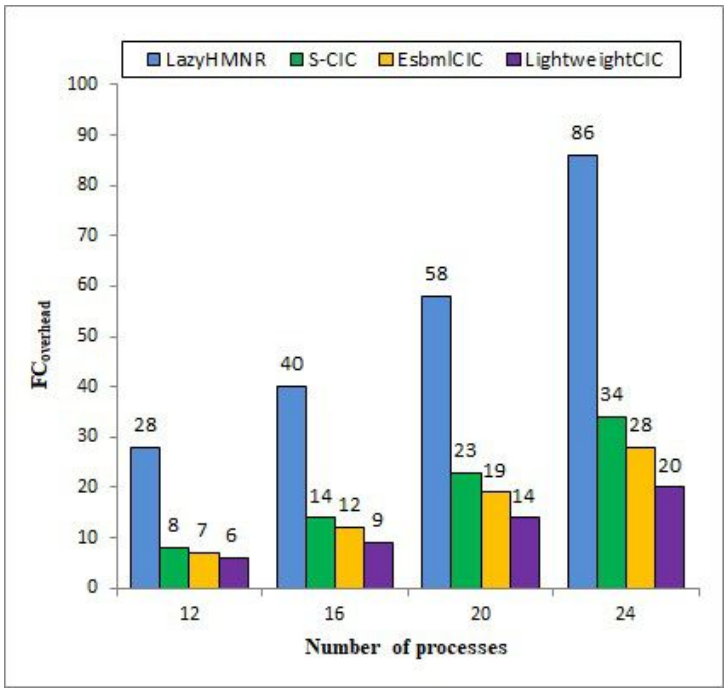

Figure 18. $FC_{overhead}$ of the four protocols in executions (UND = 0%).

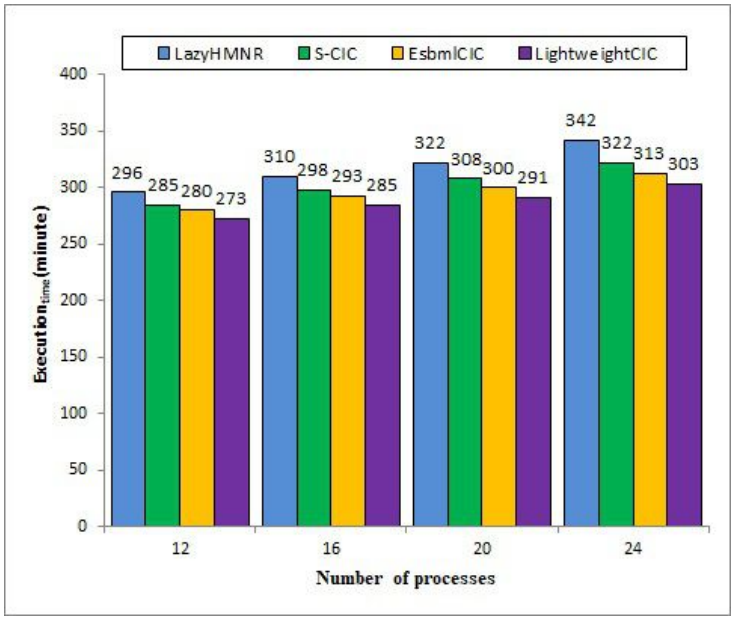

Figure 19. $Execution_{time}$ of the four protocols in executions (UND = 0%).

5. Conclusions

The proposed protocol LightweightCIC is designed to greatly reduced the forced checkpointing frequency of each process and shorten the total execution time of distributed applications by diversifying checkpoint timestamp-related information conveyance paths and greatly increasing the velocity of their propagation. For this purpose, the protocol enables each message receiver to piggyback information on the acknowledgment of each application message and intensify the bilateral up-to-dateness of the information. This behavioral feature can significantly elevate the accuracy of the awareness of situations in which forced checkpointing is actually needed for useless checkpoint-free recovery. In addition, it incurs no extra control messages and no message logging overhead while significantly reducing the latency of message sending. To verify its scalability, we proved that LightweightCIC takes fewer or the same number of forced checkpoints as HMNR [7]. Furthermore, the evaluation results in the previous section indicate the reduction rates of $FC_{overhead}$ and $Execution_{time}$ of LightweightCIC over the previous methods are in the range of 12.5% to 84.2% and 2.5% to 11.5%, in order. The overall outcome confirms our claim that our protocol can reduce the number of unnecessary forced checkpoints taken to as few as possible with its immediate update of the most recent checkpoint timestamp-related information and swift dissemination actions of the information to others.

Funding: This research was funded by Kyonggi University, grant number 2020-033.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

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

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

References

1. Garcia, I.C.; Vieira, G.M.D.; Buzato, L.E. A rollback in the history of communication-induced checkpointing. *arXiv* **2019**, arXiv:1702.06167v2.
2. Netzer, R.H.B.; Xu, J. Necessary and sufficient conditions for consistent global snapshots. *IEEE Trans. Parallel Distrib. Syst.* **1995**, *6*, 165–169. [CrossRef]
3. Wang, Y.M. Consistent global checkpoints that contain a set of local checkpoints. *IEEE Trans. Comput.* **1997**, *46*, 456–468. [CrossRef]

4. Abdelhafidi, Z.; Lagraa, N.; Yagoubi, M.B.; Djoudi, M. Low overhead communication-induced checkpointing protocols ensuring rollback-dependency trackability property. *Concurr. Comput.* **2017**, *29*, e4271. [CrossRef]
5. Acharya, A.; Badrinath, B.R. Checkpointing distributed applications on mobile computers. In Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems, Austin, TX, USA, 28–30 September 1994; pp. 73–80.
6. Manivannan, D.; Singhal, M. Quasi-synchronous checkpointing: Models, characterization, and classification. *IEEE Trans. Parallel Distrib. Syst.* **1999**, *10*, 703–713. [CrossRef]
7. 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]
8. Tsai, J. Applying the fully-informed checkpointing protocol to the lazy indexing strategy. *J. Inf. Sci. Eng.* **2007**, *23*, 1611–1621.
9. Ahn, J. Enhanced sender-based message logging for reducing forced checkpointing overhead in distributed systems. *IEICE Trans. Inf. Syst.* **2021**, *E104-D*, 1500–1505. [CrossRef]
10. Ahn, J. Communication-Induced Checkpointing with Message Logging beyond the Piecewise Deterministic (PWD) Model for Distributed Systems. *Electronics* **2021**, *10*, 1428. [CrossRef]
11. Xu, J.; Netzer, R.H.B. Adaptive independent checkpointing for reducing rollback propagation. In Proceedings of the 5th IEEE Symposium on Parallel and Distributed Processing, Dallas, TX, USA, 1–4 December 1993; pp. 754–761.
12. 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]
13. 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]
14. 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]
15. 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]
16. Alvisi, L.; Elnozahy, E.; Rao, S.; Husain, S.; de Mel, A. An analysis of communication induced checkpointing. In Proceedings of the 29th International Symposium on Fault-Tolerant Computing, Madison, WI, USA, 15–18 June 1999; pp. 242–249.
17. Jayasekara, S.; Karunasekera, S.; Harwood, A. Optimizing checkpoint-based fault-tolerance in distributed stream processing systems: Theory to practice. *Softw. Pract. Exp.* **2022**, *52*, 296–315. [CrossRef]
18. Abdelhafidi, Z.; Djoudi, M.; Lagraa, N.; Yagoubi, M.B. FNB: Fast non-blocking coordinated checkpointing protocol for distributed systems. *Theory Comput. Syst.* **2015**, *57*, 397–425. [CrossRef]
19. Vieira, G.M.D.; Buzato, L.E. Distributed checkpointing: Analysis and benchmarks. In Proceedings of the 24th Brazilian Symposium on Computer Networks, Curitiba, Brazil, 29 May–2 June 2006; pp. 1–16.
20. 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]
21. Mansouri, H.; Pathan, A. Checkpointing distributed computing systems: An optimisation approach. *Int. J. High Perform. Comput. Appl.* **2019**, *15*, 202–209. [CrossRef]
22. Sakata, T.C.; Garcia, I.C. Non-blocking synchronous checkpointing based on rollback-dependency trackability. In Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems, Leeds, UK, 2–4 October 2006; p. 411.
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]