Int. Computer Symposium, Dec. 15-17, 2004, Taipei, Taiwan.

An Algorithm for Detecting Z-cycles in Distributed Computing System

Chin-Lin Kuo and Yuo-Ming Yeh
Fault Tolerance Lab. of National Taiwan Normal University
{ *gene, ymyeh* } *@ice.ntnu.edu.tw*

Abstract- *The checkpointing approach of rollback-recovery has been widely used for fault-tolerance in distributed computing system. There are many communication messages resulting in much dependency during the time of program running. Once a process generates faults, many processes that are directly or indirectly related with the faulting process will be influenced. These processes in turn rollback to some previously stored state, respectively. What's worse, the rollback action may repeatedly trigger another rollback action of other dependent processes. This is what we know as the domino effect*[11]. *The main cause of generating domino effect is Z-cycles*[2]. *So far there is no effective method to detect Z-cycles with length more than two. In this paper, we propose a distributed algorithm to detect Z-cycles with long length.*

Keywords : fault tolerance , checkpoints , domino effect , Z-cycles , rollback-recovery.

1. Introduction

In distributed computing system, checkpointing and rollback-recovery[17] is an important mechanism for fault tolerance. A checkpoint is a stable memory record of a process state. Each process could take a checkpoint whenever process favors. The simplest solution for a process to achieve this is to take a checkpoint periodically and it will work efficiently in only one processor. But in messaging passing system with many processors, such an action are likely to generate domino effect and waste much time and computation for rollback-recovery. Every process takes checkpoints independently without considering other processes. Although this uncoordinated checkpoint method is easily implemented and allows each process to flexibly take checkpoints, it must pay much overhead, such as rollback extent, complex recovery and garbage collection.

A consistent global recovery line is a set of checkpoints, one per process, which form a recovery line. When there are faults happening on a process or processes, the process or processes in question immediately launch the rollback-recovery mechanism. If there is no valid recovery line, this action may repeatedly trigger another rollback action of other dependent processes, and the rollback distance may be unbounded and unpredictable. Many processes may have to rollback to their own initial state. This is what we call "domino effect", the worst case we would not like to encounter. In order to determine a consistent global checkpoint, the processes have to record the dependencies relation among their checkpoints during failure-free operation. However, processes cannot determine whether or not specific checkpoints are part of a consistent state.

One of the most serious problems in uncoordinated checkpointing is useless checkpoints. The processes may easily take useless checkpoints which are never part of any global consistent recovery line. Useless checkpoints are undesirable and waste much stable storage space. So applications with frequent output commits are not suitable since they could easily form many orphan messages between two checkpoints taken by two different processes and dependency relation between the states of different processes. Dependency between many processes may be occurred by message communication and there have been many papers[9,12,13] discussed about it. Another disadvantage is that determining a consistent state may be laborious and the rollback mechanism will become more complicated. Therefore most research is concentrated on coordinated checkpointing[14,15] and communication-induced checkpointing[4] schemes.

Communication-induced protocols reserves Z-cycle-free property by inserting forced checkpoints based on communication events. Hence, minimizing the number of forced checkpoints is becoming the most important topic. The main cause of generating domino effect is attributed to Z-cycles. So far, detecting Z-cycles with long length in distributed computing system is still a difficult problem. In Taesoon Park and Heon Y.Yeon's paper[3], they propose an scheme of detecting Z-cycles with length two and of taking forced checkpoints to break them under many special communication patterns. In this paper, we propose an distributed algorithm to detect all Z-cycles with long length and their involved checkpoints.

2. System Model and Background

A distributed computation consists of a finite set P of n processes $\{P_1, P_2, \cdots, P_n\}$ that interact by

means of messages sent over channels which transmission times are unpredictable but finite. Processes do not share any common memory and a common clock value, that is, they are asynchronous. The communication pattern among these processes in P could be arbitrary and the communication channel between two processes is reliable, FIFO(first-in-first-out) and bidirectional(undirectional).

Execution of a process produces a sequence of events which can be classified as: send events, receive events, and internal events. An internal statement does not involve communication. The casual ordering of events in a distributed execution is based on Lamport's *happened-before* relation[1] denoted by "$\overset{hb}{\rightarrow}$".

A process may fail, lose its volatile state and stop execution according to the fail-stop model[16]. A local checkpoint records the current process state on stable storage. The k-th checkpoint in process P_i is denoted as $C_{i,k}$, where k is an non-negative integer and we assume that each process P_i takes an initial checkpoint $C_{i,0}$ immediately before execution begins. Let $I_{i,\alpha}$ denote the interval between the consecutive checkpoints $C_{i,\alpha-1}$ and $C_{i,\alpha}$ where $\alpha = 1, 2, 3, \cdots$. In this paper, we assume each process only take local checkpoints at its own pace (for example, using a periodic algorithm) without taking forced checkpoints.

A message m sent by P_i to P_j is called an *orphan* with respect to a pair (C_{i,x_i}, C_{j,x_j}) iff its receive event happened before C_{j,x_j} while its send event happened after C_{i,x_i}. A global consistent checkpoint C is a set of local checkpoints $(C_{1,x_1}, C_{2,x_2}, \ldots, C_{n,x_n})$ which no orphan messages exists in any pair of local checkpoints belonging to C. The processes are said to rollback to the consistent recovery line if there is no orphan interval after the rollback-recovery. Sometimes, the processes have to rollback recursively to reach a consistent recovery line due to the domino effect and the rollback distance may be unbounded. In the worst case, the only consistent recovery line consists of a set of the initial checkpoints, that is, the total loss of the computation in spite of checkpointing efforts. So there are many papers talking about how to prevent domino-effect[5] or useless checkpoints[6,7].

3. Z-cycle Definition and Properties

First, we recall the Z-path definition introduced by Netzer and Xu[2].

Definition 1: *A Z-path exists from $C_{i,x}$ to $C_{j,y}$ iff there are messages m_1 , m_2 , $\cdots$, m_ℓ, $(\ell \geq 1)$ such that :*

1. *m_1 is sent by process P_i after $C_{i,x}$*
2. *if $m_k (1 \leq k < \ell)$ is received by process P_r , then m_{k+1} is sent by P_r in the same or a later checkpoint interval (although m_{k+1} may be sent before or after m_k is received).*
3. *m_ℓ is received by process P_j before $C_{j,y}$.*

Definition 2: *If there is a Z-path from $C_{i,x}$ to itself , then this is a Z-cycle which the checkpoint $C_{i,x}$ is involved.*

Assertion 1: *The length of a Z-cycle(or Z-path) is ℓ if the Z-cycle(or Z-path) is formed by ℓ messages $m_1, m_2, \cdots, m_\ell$.*

Consider some process P_i in a Z-cycle. Suppose that message m and m' are consecutive two messages contained in this Z-cycle, and message m is received by P_i and message m' is sent by the same process P_i. If $receive(m) \overset{hb}{\rightarrow} send(m')$, we say the interval between $receive(m)$ and $send(m')$ on P_i in this Z-cycle is *casual*. On the other hand, if $send(m') \overset{hb}{\rightarrow} receive(m)$, we say the interval between them is *non-casual* and they must occur in the same checkpoint interval to satisfy the definition of Z-cycle. For example, consider figure1. The Z-cycle is consisted of 4 messages m_1, m_2, m_3, m_4. On P_1, $receive(m_4) \overset{hb}{\rightarrow} send(m_1)$ so the interval is *casual*. But on P_3 , $send(m_3) \overset{hb}{\rightarrow} receive(m_2)$ and the two events occur at the same checkpoint interval so it's a *non-casual* situation. For a Z-cycle, associated with a sequence of messages $m_1, m_2, \ldots, m_\ell$, its length is ℓ and has ℓ intervals$(\ell \geq 2)$. By definition of Z-cycle, we can obtain that the interval between any two events $receive(m_i)$ and $send(m_{i+1})$ for $1 \leq i \leq \ell - 1$ has to be either a *casual* or *non-casual* interval. But there must be at least one of these intervals to be a *non-casual* interval[10]. In addition, the interval between $receive(m_\ell)$ and $send(m_1)$ must be a *casual* interval and the checkpoints between them are involved in this Z-cycle.

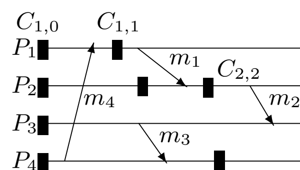

Figure 1.

Assertion 2: *For a Z-cycle, there may be more than one checkpoint involved in this Z-cycle and these checkpoints may be distributed in one or more processes.*

Obviously, the length of a Z-cycle must be at least two. In this condition, Z-cycles with length two are easy to be detected and destroied[3]. Figure 1 illustrates an example of Z-cycle with length 4 and the checkpoints $C_{1,1}$ and $C_{2,2}$ are involved in it. Intuitively, the longer Z-cycle is, the more difficult it can be detected and broken. According to Netzer and Xu's theorem, a checkpoint is said to be useless if it is involved in a Z-cycle[2], that is, it can not be included in any consistent recovery line.

4. Detecting Z-cycles Algorithm

4.1. The notation and data structures

A Z-cycle is formed by a Z-path while starting with a checkpoint and terminating at the same checkpoint. From the global view of all processes, Wang[8,9] defines a graph called the $rollback - dependency$ graph (or $R - graph$) which shows Z-paths in a distributed computation that has terminated or stopped execution. It is easy to find Z-paths from such a graph. In distributed algorithm, each process only has its local memory and knows the (send and receive) events relative to itself but does not know other messages' transmission in other processes. Hence a process may not have ability to accumulate sufficient information of message transmission to concatenate them into Z-paths without piggybacked information. So the most critical problem to detect Z-cycles is how to collect necessary messages $m_1, m_2, \cdots, m_\ell$ which may have any possibility of forming a Z-cycle. First, we have to conceptualize an appropriate data structure to express Z-path and Z-cycle.

For a single message m, its important four characteristics are the two processes which send, receive m and the two checkpoint intervals while the sending, receiving events occurring. There are totally four natural numbers, $send_Pid$, ℓc_{out} on process send-Pid, $receive_Pid$, and ℓc_{in} on process receive-Pid to describe the message m. For example, if there is a message m which was sent by process P_i in checkpoint interval $I_{i,\alpha}$ and received by process P_j in $I_{j,\beta}$, then $send_Pid = i$, $receive_Pid = j$, ℓc_{out} in P_i is α and ℓc_{in} in P_j is β. We use the symbol $[\underset{\square,\alpha}{i}, \underset{\beta,\square}{j}]$ to express m. The lower-left $\square$ of i and lower-right $\square$ of j mean a checkpoint interval number of a message delivery event in P_i and a checkpoint interval number of another message sending event occurring in P_j respectively. These two $\square$s are written out for the purpose of connecting messages to form a Z-path.

Notation : *A message m which is sent by P_i in $I_{i,\alpha}$ and received by P_j in $I_{j,\beta}$ is denoted by $[\underset{\square,\alpha}{i}, \underset{\beta,\square}{j}]$. The symbol $\square$ means $unknown$ or not $occurred$ yet and α, β are natural numbers.*

This notation of a single message can completely express relative information in a Z-path and from that we can only pay attention to the notation instead of $R - graph$.

Lemma 1: *For a process P_j, if there are two messages m_1, m_2, which are denoted by $[\underset{\square,\alpha}{i}, \underset{\beta,\square}{j}]$ and $[\underset{\square,\gamma}{j}, \underset{\theta,\square}{k}]$ respectively ,where $\alpha, \beta, \gamma, \theta \in N$(natural number), then we check whether $\beta \leq \gamma$. If $\beta \leq \gamma$ holds, then the second condition of Z-path's definition is satisfied and so we can merge(connect) these two messages into a Z-path, represented by $[\underset{\square,\alpha}{i}, \underset{\beta,\gamma}{j}, \underset{\theta,\square}{k}]$*

proof : These two messages m_1, m_2, denoted by $[\underset{\square,\alpha}{i}, \underset{\beta,\square}{j}]$ and $[\underset{\square,\gamma}{j}, \underset{\theta,\square}{k}]$ respectively ,where $\alpha, \beta, \gamma, \theta \in N$, mean that P_i sends m_1 in $I_{i,\alpha}$ to P_j in $I_{j,\beta}$ and P_j sends m_2 in $I_{j,\gamma}$ to P_k in $I_{k,\theta}$. When $\beta = \gamma$, it means m_1 is received by P_j and m_2 is sent by P_j in the same checkpoint interval no matter $receive(m_1) \overset{hb}{\rightarrow} send(m_2)$ or $send(m_2) \overset{hb}{\rightarrow} receive(m_1)$. The interval between the two events probably could be casual or non-casual. When $\beta < \gamma$, it means $receive(m_1) \overset{hb}{\rightarrow} send(m_2)$ and $send(m_2)$ occurs in a later checkpoint interval. So by the second condition of Z-path's definition, if one of the above two conditions($\beta = \gamma$ or $\beta < \gamma$) holds, then m_1 and m_2 could be merged into a Z-path $[\underset{\square,\alpha}{i}, \underset{\beta,\gamma}{j}, \underset{\theta,\square}{k}]$. But if $\beta > \gamma$, m_1 and m_2 could not be merged since these two checkpoint intervals $I_{j,\beta}$ and $I_{j,\gamma}$ contradict the definition 2 of Z-path. ⊠

From above discussion, the length of a Z-path can gradually increase by merging messages one by one or merging other Z-paths. Contrarily, a Z-path $[\cdots, \underset{\cdots,\alpha}{i}, \underset{\beta,\gamma}{j}, \underset{\theta,\cdots}{k}, \cdots]$ could be decomposed into two Z-paths, $[\cdots, \underset{\cdots,\alpha}{i}, \underset{\beta,\square}{j}]$ and $[\underset{\square,\gamma}{j}, \underset{\theta,\cdots}{k}, \cdots]$. The rules of merging two Z-paths $path1$ and $path2$ are to check (1) whether the last Pid of $path1$ is equal to the first Pid of $path2$ and (2) whether the ℓc_{in} of the last Pid of $path1$ is equal to or smaller than the ℓc_{out} of the first P_id of $path2$. If satisfied, then these two Z-paths could be merged into a single Z-path $[\cdots, \underset{\cdots,\alpha}{i}, \underset{\beta,\gamma}{j}, \underset{\theta,\cdots}{k}, \cdots]$. We use notation $[\underset{\square,b_1}{1}, \underset{a_2,b_2}{2}, \underset{a_3,b_3}{3}, \cdots, \underset{a_n,b_n}{n}, \underset{a_k,\square}{k}]$ to express a Z-path from checkpoint C_{1,b_1-1} to C_{k,a_k}. Certainly a_i, b_i are natural numbers and the relation $a_i \leq b_i$ must holds for every process. If $k = 1$ and $a_k \leq b_1$ then Z-cycle $(\underset{a_1,b_1}{1}, \underset{a_2,b_2}{2}, \underset{a_3,b_3}{3}, \cdots, \underset{a_n,b_n}{n})$ forms.

The length of a Z-path is not fixed, so for data structure representation, the way of utilizing queue can appropriately express the meaning of Z-path. Each element of the queue has three integers Pid, ℓc_in and ℓc_out, where $Pid \in \{1, 2, \ldots, n\}$ means process ID and ℓc_in,ℓc_out means the checkpoint interval $I_{Pid,\ell c_in}$ of the $receive$ event and the checkpoint interval $I_{Pid,\ell c_out}$ of the $send$ event on the same process Pid respectively.

Assertion 3: The data structure "queue of Z-path" we define can appropriately express the meaning of Z-path.

Assertion 4: For a Z-path $[\cdots, \underset{\alpha,\beta}{i}, \cdots]$, where $\alpha \leq \beta$, α and β means the checkpoint interval $I_{i,\alpha}$ of event $receive(m_s)$ and $I_{i,\beta}$ of event $send(m_t)$ respectively for some $s, t \in N$. If $\alpha = \beta$, then these

two events, $receive(m_s)$ and $send(m_t)$, occur in the same checkpoint interval. If $\alpha < \beta$, then there are $\beta - \alpha$ checkpoints $C_{i,\alpha}, C_{i,\alpha+1}, \cdots, C_{i,\beta-1}$ between $receive(m_s)$ and $send(m_t)$. For the case $\alpha < \beta$, if the Z-path can form a Z-cycle in the future, then the checkpoints $C_{i,\alpha}, C_{i,\alpha+1}, \cdots, C_{i,\beta-1}$ are involved in this Z-cycle. For example, in figure 1 there is a Z-cycle $(\underset{1,1}{4}, \underset{1,2}{1}, \underset{2,3}{2}, \underset{1,1}{3})$ in which checkpoints $\{C_{1,1}, C_{2,2}\}$ are involved.

The following paragraph lists the notations and data structures used in our algorithm. There are n processes and for each process P_i it has

- ℓc_i : an integer and a logical counter which means current checkpoint interval index between two consecutive checkpoints and its initial value is 1.

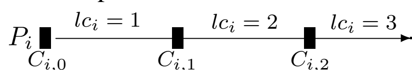

Figure 2.

- Z_Queue_i : A queue which each element of it is still a queue z_path containing Z-path information, for example $[\underset{\square,2}{4}, \underset{2,2}{2}, \underset{1,\square}{1}]$. In a node of z_path, there are three integers which mean process's id Pid and its two subscripts below, ℓc_in and ℓc_out. If one of them are $\square$, it means $unknown$, which could only appear at the ℓc_in of the first Pid and the ℓc_out of the last Pid in a Z-path. The $[\underset{\square,\cdots}{1}, \underset{\cdots,\cdots}{2}, \underset{\cdots,\cdots}{3}, \ldots]$ means Z-path from process P_1 to P_2 , P_3 , $\cdots\cdots$. The ℓc_in is smaller or equal to the ℓc_out. Maybe there are many Z-paths included in the Z_Queue_i. Its structure is as the following figure and its initial value is $null$.

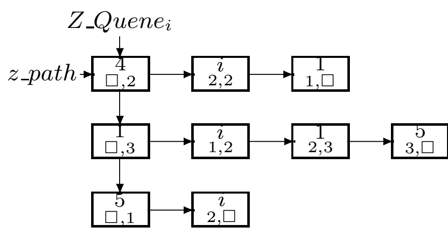

Figure 3.

- $Z_Queue_buffer1_i$: A Z-path queue buffer which stores the Z-path queue piggybacked from other processes and is used to merge them with its own Z_Queue_i.
- $Z_Queue_buffer2_i$: A Z-path queue buffer which also stores a queue of Z-paths. If P_i needs to send *z-path request* message to other processes, then P_i must wait to receive for replying *z-path*s from them and store these z_paths into $Z_Queue_buffer2_i$.
- csn_i : checkpoint line which is an array of n checkpoint sequence numbers(csn) and $csn_i[j]$ represents the largest checkpoint sequence number of P_j that P_i knows. The value of $csn_i[i]$ is always equal to $(\ell c_i - 1)$. Its initial value is $[0, 0, \cdots, 0]$.
- Z_cycle_i : An Z-cycle list which each element stores a Z-cycle. Initial values are $none$

4.2. The algorithm

We distinguish two kinds of messages: *computation messages* and *system messages*. Computation messages are sent for their application purposes. In our protocol there are two kinds of system message, "*z-path request*" and "*z-path reply*". This algorithm mainly adopt piggyback approach and request Z-paths from other processes to accumulate sufficient information. Then process merges its own Z-paths with them to check whether Z-cycles form or not. Not every time P_i has to send *z-path request* to collect another process's Z-paths. When there were sending events occurred after the latest checkpoint in P_i and the P_i receives a computation message (non-casual), P_i needs to do so. By the definition of Z-cycle formed by $m_1, m_2, \cdots, m_\ell$, the checkpoint interval between m_1 and m_ℓ must be casual and there must exist at least one non-casual interval[10] in a Z-cycle. For our algorithm, the less number of non-casual intervals, the more efficient performance we have. So there are briefly three different cases of Z-cycles(best, worst, average). The figures 4,5 and 1 illustrate the three situations.

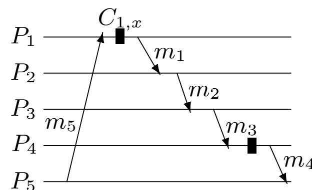

Figure 4 : best case

For the best case like the above figure 4, there is only one *non-casual* interval(between m_5 and m_4) in the Z-cycle. When P_5 receives m_4, it checks there is a computation message m_5 sent to P_1 in the current checkpoint interval. So, P_5 must send a *z-path request* message to P_1 for obtaining $[\underset{\square,\cdots}{5}, \underset{\cdots,\cdots}{1}, \cdots]$. In the best case, most of these intervals are *casual* and only few processes need to send *z-path request* for more Z-path information.

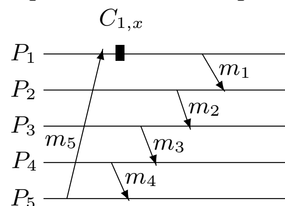

Figure 5 : worst case

But in the worst case like figure 5, most of the intervals are *non-casual*. Hence most processes(P_5, P_4, P_3, P_2) have to send *z-path request* to other processes for more Z-path information. The performance would be decreased. Each time of computation message-passing occurring the message must bring many Z-paths data, which may be tremendous, to target process and then the target process connects these received Z-paths data with its own. There will generate many new Z-paths in the connecting action and Z-cycle(s) will be detected. The following part is the explanation of our algorithm.

Sending a computation message: P_i sends a computation message to P_j. Let the computation message be denoted by $[\underset{\Box,\ell c_i}{i}, \underset{\Box,\Box}{j}]$. For each Z-path in Z_Queue_i we only duplicate the front part of the Z-path, $[\cdots\cdots, \underset{\alpha,\Box}{i}]$, for some α, to merge with $[\underset{\Box,\ell c_i}{i}, \underset{\Box,\Box}{j}]$.Then P_i obtains a new Z-path $[\cdots\cdots, \underset{\alpha,\ell c_i}{i}, \underset{\Box,\Box}{j}]$, where $\alpha \leq \ell c_i$. There probably are many such new Z-paths and all of them piggyback the computation message forwarding to P_j.

Reception of a computation message and piggybacked information: When P_i receives a computation message M and piggybacked information(Z-paths) from P_k, each of them as $[\cdots, \underset{\cdots,\cdots}{k}, \underset{\Box,\Box}{i}]$, the first step P_i must do is to write ℓc_i into them, $[\cdots, \underset{\cdots,\cdots}{k}, \underset{\ell c_i,\Box}{i}]$. P_i can update csn_i by these piggybacked Z-paths. That is, P_i can move checkpoint line forward to the latest checkpoint index which P_i can know. After updating csn_i, P_i can also prune these piggybacked Z-paths. In Z_Queue_i if there exists Z-paths like $[\cdots, \underset{\cdots,\ell c_i}{i}, \underset{\Box,\Box}{j}]$, which means there is a computation message sending from P_i to P_j in the current checkpoint interval of index ℓc_i, then P_i has to send a *z-path request* for P_j in order to obtain sufficient information of Z-path as $[\underset{\Box,\ell c_i}{i}, \underset{\cdots,\cdots}{j}, \cdots]$. Then P_i can connect $[\cdots, \underset{\cdots,\cdots}{k}, \underset{\cdots,\ell c_i}{i}, \underset{\Box,\Box}{j}]$ with $[\underset{\Box,\ell c_i}{i}, \underset{\cdots,\cdots}{j}, \cdots]$ into $[\cdots, \underset{\cdots,\cdots}{k}, \underset{\cdots,\ell c_i}{i}, \underset{\cdots,\cdots}{j}, \cdots]$. If there is any Z-cycle formed due to the message $[\underset{\Box,\cdots}{k}, \underset{\ell c_i,\Box}{i}]$, then we can detect the Z-cycle containing it.

Procedure PruneZ-path(csn_i,Z_Queue_i): The data of csn_i in P_i means the checkpoint line that P_i already knows. When the csn_i is updated, P_i checks each Z-path in Z_Queue_i whether its ℓc_out of first Pid is equal to or smaller than $csn_i[Pid]$. That is, the event $send(m)$ of the first message m in the Z-path occurred before checkpoint $C_{Pid,csn_i[Pid]}$, the left side of the checkpoint line csn_i. If so, it implies that there could not be any messages received by P_{Pid} at that checkpoint interval in the future. Then the first message of the Z-path should be deleted. Repeat such pruning action till the ℓc_out of first Pid in this Z-path is larger than $csn_i[Pid]$.

When P_i receives a *z-path request*($[\underset{\Box,\ell c_q}{q}, \underset{\Box,\Box}{i}]$) from P_q: If P_i receives such *z-path request* and its parameter $[\underset{\Box,\ell c_q}{q}, \underset{\Box,\Box}{i}]$, it means that there was a computation message sent by P_q to P_i. But P_q doesn't know the checkpoint interval index of the computation message arrived at P_i. For P_i there must be a Z-path $[\cdots, \underset{\cdots,\ell c_q}{q}, \underset{\alpha,\beta}{i}, \cdots]$ in Z_Queue_i, for some α, β. We duplicate the back part, $[\underset{\Box,\ell c_q}{q}, \underset{\alpha,\beta}{i}, \cdots]$ and reply them for P_q. After collecting such Z-paths, $[\underset{\Box,\ell c_q}{q}, \underset{\alpha,\beta}{i}, \cdots]$, P_q can connect them with its own Z-paths, $[\cdots, \underset{\cdots,\Box}{q}]$. So P_q can check whether Z-cycles form or not. We demonstrate our algorithm by an example.

Example : In this example figure 6, there are totally two Z-cycles,$\{m_3, m_5, m_1\}$ and $\{m_4, m_3, m_5, m_2\}$. The checkpoints involved are $\{C_{2,1}, C_{3,2}\}$ and $\{C_{1,2}, C_{3,2}\}$ respectively. So we can observe that messages m_3 and m_5 are associated with these two Z-cycles simultaneously.

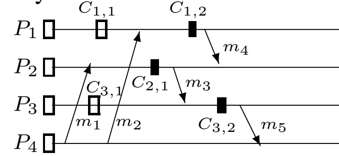

Figure 6.

We illustrate this example by the order of messages occurring time and present the csn and Z_Queue data of Z-paths for all processes at the time of sending, receiving and checkpointing. The concatenation of two Z-paths is expressed by $path_1 + path_2 \Rightarrow \cdots$.

$send(m_1)$:
P_1: csn_1 : (0000) ; empty
P_2: csn_2 : (0000) ; empty
P_3: csn_3 : (0000) ; empty
P_4: csn_4 : (0000) ; $[\underset{\Box,1}{4}, \underset{\Box,\Box}{2}]$

$receive(m_1)$: $[\underset{\Box,1}{4}, \underset{1,\Box}{2}]$ piggybacked to P_2
P_1: csn_1 : (0000) ; empty
P_2: csn_2 : (0000) ; $[\underset{\Box,1}{4}, \underset{1,\Box}{2}]$
P_3: csn_3 : (0000) ; empty
P_4: csn_4 : (0000) ; $[\underset{\Box,1}{4}, \underset{\Box,\Box}{2}]$

P_3 takes $C_{3,1}$, csn_3 : (0010); empty
P_1 takes $C_{1,1}$, csn_1 : (1000); empty

$send(m_2)$:
P_1: csn_1 : (1000) ; empty
P_2: csn_2 : (0000) ; $[\underset{\Box,1}{4}, \underset{1,\Box}{2}]$
P_3: csn_3 : (0010) ; empty
P_4: csn_4 : (0000) ; $[\underset{\Box,1}{4}, \underset{\Box,\Box}{2}]$ and $[\underset{\Box,1}{4}, \underset{\Box,\Box}{1}]$

$receive(m_2)$: $[\underset{\Box,1}{4}, \underset{2,\Box}{1}]$ piggybacked to P_1

P_1: csn_1 : (1000) ; $[\underset{\square,1}{4},\underset{2,\square}{1}]$
P_2: csn_2 : (0000) ; $[\underset{\square,1}{4},\underset{1,\square}{2}]$
P_3: csn_3 : (0010) ; empty
P_4: csn_4 : (0000) ; $[\underset{\square,1}{4},\underset{\square,\square}{2}]$ and $[\underset{\square,1}{4},\underset{\square,\square}{1}]$
P_2 takes $C_{2,1}$, csn_2 : (0100) ; $[\underset{\square,1}{4},\underset{1,\square}{2}]$
$send(m_3)$:
P_1: csn_1 : (1000) ; $[\underset{\square,1}{4},\underset{2,\square}{1}]$
P_2: csn_2 : (0100) ; $[\underset{\square,1}{4},\underset{1,\square}{2}]$ + $[\underset{\square,2}{2},\underset{\square,\square}{3}]$ $\Rightarrow$ $[\underset{\square,1}{4},\underset{1,2}{2},\underset{\square,\square}{3}]$ piggybacked to P_3
P_3: csn_3 : (0010) ; empty
P_4: csn_4 : (0000) ; $[\underset{\square,1}{4},\underset{\square,\square}{2}]$ and $[\underset{\square,1}{4},\underset{\square,\square}{1}]$
P_1 takes $C_{1,2}$, csn_1 : (2000) ; $[\underset{\square,1}{4},\underset{2,\square}{1}]$
$receive(m_3)$: $[\underset{\square,1}{4},\underset{1,2}{2},\underset{2,\square}{3}]$ piggybacked to P_3
P_1: csn_1 : (2000) ; $[\underset{\square,1}{4},\underset{2,\square}{1}]$
P_2: csn_2 : (0100) ; $[\underset{\square,1}{4},\underset{1,2}{2},\underset{\square,\square}{3}]$
P_3: Update csn_3 : (0110) ; $[\underset{\square,1}{4},\underset{1,2}{2},\underset{2,\square}{3}]$
P_4: csn_4 : (0000) ; $[\underset{\square,1}{4},\underset{\square,\square}{2}]$ and $[\underset{\square,1}{4},\underset{\square,\square}{1}]$
$send(m_4)$:
P_1: csn_1 : (2000) ; $[\underset{\square,1}{4},\underset{2,\square}{1}]$ + $[\underset{\square,3}{1},\underset{\square,\square}{2}]$ $\Rightarrow$ $[\underset{\square,1}{4},\underset{2,3}{1},\underset{\square,\square}{2}]$ which piggybacks m_4
P_2: csn_2 : (0100) ; $[\underset{\square,1}{4},\underset{1,2}{2},\underset{\square,\square}{3}]$
P_3: csn_3 : (0110) ; $[\underset{\square,1}{4},\underset{1,2}{2},\underset{2,\square}{3}]$
P_4: csn_4 : (0000) ; $[\underset{\square,1}{4},\underset{\square,\square}{2}]$ and $[\underset{\square,1}{4},\underset{\square,\square}{1}]$
$receive(m_4)$: $[\underset{\square,1}{4},\underset{2,3}{1},\underset{2,\square}{2}]$ piggybacked to P_2
P_1: csn_1 : (2000) ; $[\underset{\square,1}{4},\underset{2,3}{1},\underset{\square,\square}{2}]$
P_2: Update csn_2 : (2100) ; sends $request([\underset{\square,2}{2},\underset{\square,\square}{3}])$ to P_3 to get $[\underset{\square,2}{2},\underset{2,\square}{3}]$.
So $[\underset{\square,1}{4},\underset{2,3}{1},\underset{2,\square}{2}] + [\underset{\square,2}{2},\underset{2,\square}{3}] \Rightarrow [\underset{\square,1}{4},\underset{2,3}{1},\underset{2,2}{2},\underset{2,\square}{3}]$
and $[\underset{\square,1}{4},\underset{1,2}{2},\underset{\square,\square}{3}] + [\underset{\square,2}{2},\underset{2,\square}{3}] \Rightarrow [\underset{\square,1}{4},\underset{1,2}{2},\underset{2,\square}{3}]$
P_3: csn_3 : (0110) ; $[\underset{\square,1}{4},\underset{1,2}{2},\underset{2,\square}{3}]$
P_4: csn_4 : (0000) ; $[\underset{\square,1}{4},\underset{\square,\square}{2}]$ and $[\underset{\square,1}{4},\underset{\square,\square}{1}]$
P_3 takes $C_{3,2}$, csn_3 : (0120). $[\underset{\square,1}{4},\underset{1,2}{2},\underset{2,\square}{3}]$
$send(m_5)$:
P_1: csn_1 : (2000) ; $[\underset{\square,1}{4},\underset{2,3}{1},\underset{\square,\square}{2}]$
P_2: csn_2 : (2100) ; $[\underset{\square,1}{4},\underset{2,3}{1},\underset{2,2}{2},\underset{2,\square}{3}]$
and $[\underset{\square,1}{4},\underset{1,2}{2},\underset{2,\square}{3}]$
P_3: csn_3 : (0120) ; $[\underset{\square,1}{4},\underset{1,2}{2},\underset{2,\square}{3}]$ + $[\underset{\square,3}{3},\underset{\square,\square}{4}]$ $\Rightarrow$ $[\underset{\square,1}{4},\underset{1,2}{2},\underset{2,3}{3},\underset{\square,\square}{4}]$ which piggybacks m_5
P_4: csn_4 : (0000) ; $[\underset{\square,1}{4},\underset{\square,\square}{2}]$ and $[\underset{\square,1}{4},\underset{\square,\square}{1}]$

$receive(m_5)$: $[\underset{\square,1}{4},\underset{1,2}{2},\underset{2,3}{3},\underset{1,\square}{4}]$ piggybacked to P_4
P_1:csn_1 : (2000);$[\underset{\square,1}{4},\underset{2,3}{1},\underset{\square,\square}{2}]$
P_2:csn_2 : (2100);$[\underset{\square,1}{4},\underset{2,3}{1},\underset{2,2}{2},\underset{2,\square}{3}]$ and $[\underset{\square,1}{4},\underset{1,2}{2},\underset{2,\square}{3}]$
P_3: csn_3 : (0120) ; $[\underset{\square,1}{4},\underset{1,2}{2},\underset{2,3}{3},\underset{\square,\square}{4}]$
P_4: Update csn_4 : (0120) ; $[\underset{\square,1}{4},\underset{1,2}{2},\underset{2,3}{3},\underset{1,\square}{4}]$+$[\underset{\square,1}{4},\underset{\square,\square}{2}]$
$\Rightarrow[\underset{\square,1}{4},\underset{1,2}{2},\underset{2,3}{3},\underset{1,1}{4},\underset{\square,\square}{2}]$
$[\underset{\square,1}{4},\underset{1,2}{2},\underset{2,3}{3},\underset{1,\square}{4}]$+$[\underset{\square,1}{4},\underset{\square,\square}{1}]$$\Rightarrow$$[\underset{\square,1}{4},\underset{1,2}{2},\underset{2,3}{3},\underset{1,1}{4},\underset{\square,\square}{1}]$
Since there are $[\cdots,\underset{\square,\square}{2}]$ and $[\cdots,\underset{\square,\square}{1}]$, P_4 sends $request([\underset{\square,1}{4},\underset{\square,\square}{2}])$ to P_2 to get $[\underset{\square,1}{4},\underset{1,2}{2},\underset{2,\square}{3}]$.
P_4 sends $request([\underset{\square,1}{4},\underset{\square,\square}{1}])$ to P_1.When P_1 receives the $request$,P_1 plans to reply $[\underset{\square,1}{4},\underset{2,3}{1},\underset{\square,\square}{2}]$.But there is $[\cdots,\underset{\square,\square}{2}]$, P_1 has to send $request([\underset{\square,3}{1},\underset{\square,\square}{2}])$ to P_2 to get $[\underset{\square,3}{1},\underset{2,2}{2},\underset{2,\square}{3}]$. In P_1, $[\underset{\square,1}{4},\underset{2,3}{1},\underset{\square,\square}{2}]$+$[\underset{\square,3}{1},\underset{2,2}{2},\underset{2,\square}{3}]$$\Rightarrow$$[\underset{\square,1}{4},\underset{2,3}{1},\underset{2,2}{2},\underset{2,\square}{3}]$. So P_1 replies $[\underset{\square,1}{4},\underset{2,3}{1},\underset{2,2}{2},\underset{2,\square}{3}]$ for P_4's $request$.
Then P_4 has the following action :
$[\underset{\square,1}{4},\underset{1,2}{2},\underset{2,3}{3},\underset{1,1}{4},\underset{\square,\square}{2}]$+$[\underset{\square,1}{4},\underset{1,2}{2},\underset{2,\square}{3}]$
$\Rightarrow$ $[\underset{\square,1}{4},\underset{1,2}{2},\underset{2,3}{3},\underset{1,1}{4},\underset{1,2}{2},\underset{2,\square}{3}]$
$[\underset{\square,1}{4},\underset{1,2}{2},\underset{2,3}{3},\underset{1,1}{4},\underset{\square,\square}{1}]$+$[\underset{\square,1}{4},\underset{2,3}{1},\underset{2,2}{2},\underset{2,\square}{3}]$
$\Rightarrow$ $[\underset{\square,1}{4},\underset{1,2}{2},\underset{2,3}{3},\underset{1,1}{4},\underset{2,3}{1},\underset{2,2}{2},\underset{2,\square}{3}]$. So Z-cycles $(\underset{1,1}{4},\underset{1,2}{2},\underset{2,3}{3})$,$(\underset{2,2}{2},\underset{2,3}{3},\underset{1,1}{4},\underset{2,3}{1})$ will be detected and involved checkpoints are $\{C_{2,1},C_{3,2}\}$, $\{C_{3,2},C_{1,2}\}$ respectively.

5. Proof of correctness

5.1. Theorem : *Our algorithm can detect all Z-cycles in distributed computing system.*

For Z-cycle detection algorithm, the crucial question is that a process should accumulate necessary and sufficient information of messages passing and merge these data to check Z-cycle.

proof : Without losing generality, we assume there is a Z-cycle associated with a sequence of messages $m_1, m_2, \ldots, m_\ell$ and the representation of the Z-cycle is $[\underset{\square,b_1}{1},\underset{a_2,b_2}{2},\cdots,\underset{a_\ell,b_\ell}{\ell},\underset{a_1,\square}{1}]$, where $\ell \geq 2$. We prove this theorem by induction on the length ℓ of Z-cycle.
When $\ell = 2$, the figure of such Z-cycle is as figure 7.

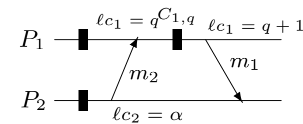

Figure 7.

For P_2, when m_2 is sent to P_1 at $I_{2,\alpha}$, m_2 = $[\underset{\square,\alpha}{2},$

$\underset{\square,\square}{1}]$ is placed in Z_Queue_2. Till P_1 receives m_2 from P_2, m_2 is piggybacked to P_1 and P_1 can fill $\ell c_1 = q$ value into m_2, that is, $m_2 = [\underset{\square,\alpha}{2}, \underset{q,\square}{1}]$ in Z_Queue_1. After P_1 taking a checkpoint $C_{1,q}$, P_1 sends $m_1 = [\underset{\square,q+1}{1}, \underset{\square,\square}{2}]$ to P_2. Before the sending event, P_1 merges m_1 with Z_Queue_1 and then there will be a Z-path $[\underset{\square,\alpha}{2}, \underset{q,q+1}{1}, \underset{\square,\square}{2}]$ generated in Z_Queue_1. When m_1 arrives P_2, it piggybacks the Z-path to P_2 and so P_2 can fill $\ell c_2 = \alpha$ into the lower-left $\square$ of $\underset{\square,\square}{2}$. Then there is a Z-path $[\underset{\square,\alpha}{2}, \underset{q,q+1}{1}, \underset{\alpha,\square}{2}]$ contained in Z_Queue_2. So P_2 can detect the Z-cycle $[\underset{\square,\alpha}{2}, \underset{q,q+1}{1}, \underset{\alpha,\square}{2}]$,that is $(\underset{\alpha,\alpha}{2}, \underset{q,q+1}{1})$. By this notation we can also induct that the checkpoint $C_{1,q}$ is involved in this Z-cycle.

Suppose when $\ell = k$, the theorem is true. That is, a Z-cycle associated with k messages m_1 , m_2 , $\cdots$, m_k denoted by $[\underset{\square,b_1}{1}, \underset{a_2,b_2}{2}, \cdots, \underset{a_k,b_k}{k}, \underset{a_1,\square}{1}]$ can be detected at process P_i, for some i.

Then when $\ell = k+1$, we must show a Z-cycle associated with $k+1$ messages $m_1, m_2, \cdots, m_k, m_{k+1}$ could be detected at some process. Let m_1 and m_k be the neighbor messages of m_{k+1} and the Z-cycle is $\{\cdots, m_k, m_{k+1}, m_1, \cdots\}$. According to the time of events m_k, m_{k+1}, m_1 occurring, there are four cases. Assume P_s receives m_k and sends m_{k+1} , and P_t receives m_{k+1} and sends m_1 to P_r.

case I : For P_s, $receive(m_k) \stackrel{hb}{\rightarrow} send(m_{k+1})$ and on P_t, $receive(m_{k+1}) \stackrel{hb}{\rightarrow} send(m_1)$. That is, P_s is *casual* and P_t is also *casual*.

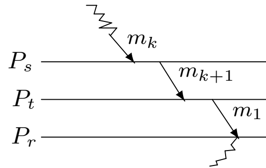

Figure 8.

For P_s, when P_s receives m_k, it contains $[\underset{\square,b_1}{1}, \cdots, \underset{a_s,\square}{s}]$ in Z_Queue_s. Since $receive(m_k) \stackrel{hb}{\rightarrow} send(m_{k+1})$, so when the event $send(m_{k+1})$ occurs, the Z-path will be merged with $[\underset{\square,b_s}{s}, \underset{\square,\square}{t}]$ and then becomes $[\underset{\square,b_1}{1}, \cdots, \underset{a_s,b_s}{s}, \underset{\square,\square}{t}]$ which will be piggybacked to P_t. For P_t, it receives $[\underset{\square,b_1}{1}, \cdots, \underset{a_s,b_s}{s}, \underset{\square,\square}{t}]$ and it can fill $a_t = \ell c_t$ into the lower-left $\square$ of P_t. Since $receive(m_{k+1}) \stackrel{hb}{\rightarrow} send(m_1)$, so when P_t sends m_1,denoted by $[\underset{\square,\ell c_t}{t}, \underset{\square,\square}{r}]$, $[\underset{\square,b_1}{1}, \cdots, \underset{a_s,b_s}{s}, \underset{a_t,b_t}{t}, \underset{\square,\square}{r}]$, where $b_t = \ell c_t$, will be piggybacked to its target process P_r. For P_r, when P_r receives m_1, it can have $[\underset{\square,b_1}{1}, \cdots, \underset{a_s,b_s}{s}, \underset{a_t,b_t}{t}, \underset{a_r,\square}{r}]$ in Z_Queue_r. So by our algorithm, the message m_{k+1} could be completely inserted into the Z-cycle which could be detected.

case II : On P_s, $receive(m_k) \stackrel{hb}{\rightarrow} send(m_{k+1})$ and on P_t, $send(m_1) \stackrel{hb}{\rightarrow} receive(m_{k+1})$. That is, P_s is *casual* and P_t is *non-casual*.

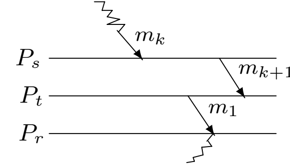

Figure 9.

By case I, when P_t receives m_{k+1}, Z_Queue_t contains $[\underset{\square,b_1}{1}, \cdots, \underset{a_s,b_s}{s}, \underset{a_t,\square}{t}]$, where $a_t = \ell c_t$. But m_1 has already been sent, so Z_Queue_t contains $[\underset{\square,a_t}{t}, \underset{\square,\square}{r}]$ and after merge action, Z_Queue_t will generate $[\underset{\square,b_1}{1}, \cdots, \underset{a_s,b_s}{s}, \underset{a_t,a_t}{t}, \underset{\square,\square}{r}]$, in which there are two $\square$ symbols at process r. So Pt will send a *request* message for Z_Queue_r to obtain $[\underset{\square,a_t}{t}, \underset{a_r,\cdots}{r}, \cdots]$ from Pr. And then Pt merges again, Z_Queue_t will get $[\cdots, \underset{a_s,b_s}{s}, \underset{a_t,a_t}{t}, \underset{a_r,b_r}{r}, \cdots]$. So the message m_{k+1} could be also inserted into the Z-cycle.

case III : For P_s, $send(m_{k+1}) \stackrel{hb}{\rightarrow} send(m_k)$ and for P_t, $receive(m_{k+1}) \stackrel{hb}{\rightarrow} send(m_1)$. That is, P_s is *non-casual* and P_t is *casual*.

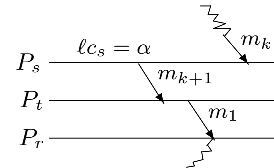

Figure 10.a

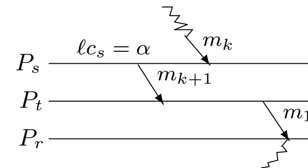

Figure 10.b

For P_s, since $send(m_{k+1}) \stackrel{hb}{\rightarrow} receive(m_k)$, so Z_Queue_s contains $[\cdots, \underset{\square,\alpha}{s}, \underset{\square,\square}{t}]$. When P_s receives m_k, Z_Queue_s will contains $[\cdots, \underset{\alpha,\square}{s}]$. So they will be merged into $[\cdots, \underset{\alpha,\alpha}{s}, \underset{\square,\square}{t}]$. Because there are two $\square$ symbols in P_t, P_s will send P_t a *request* and then merges with Z_Queue_s to obtain $[\cdots, \underset{\alpha,\alpha}{s}, \underset{\beta,\square}{t}]$. For P_t, when P_t receives m_{k+1}, Z_Queue_t contains $[\underset{\square,\alpha}{s}, \underset{\beta,\square}{t}]$, for some *beta*. Later when P_t sends m_1 to P_r, Z_Queue_t contains $[\underset{\square,\beta'}{t}, \underset{\square,\square}{r}]$, where $\beta' \geq \beta$. So

P_t could have $[s_{\square,\alpha}, t_{\beta,\beta'}, r_{\square,\square}]$. When P_r receives m_1, $[s_{\square,\alpha}, t_{\beta,\beta'}, r_{\theta,\square}]$, for some θ, will be obtained. As figure 10, there are two distinct situations.

If $receive(m_1) \xrightarrow{hb} receivr(m_k)$, as figure 10.a, then Z_Queue_s has $[\cdots, s_{\alpha,\square}]$ and $[s_{\square,\alpha}, t_{\beta,\beta'}, r_{\theta,\cdots}, \cdots]$ after *requesting* P_t. So P_s could obtain $[\cdots, s_{\alpha,\alpha}, t_{\beta,\beta'}, r_{\theta,\cdots}, \cdots]$.

If $receive(m_k) \xrightarrow{hb} receivr(m_1)$, as figure 10.b, then Z_Queue_r has $[s_{\square,\alpha}, t_{\beta,\beta'}, r_{\theta,\square}]$. When P_r receives m_1, it sends a *request* for some process P_u to get $[u_{\square,\cdots}, \cdots, s_{\alpha,\square}]$. So after connection, P_r could obtain $[u_{\square,\cdots}, \cdots, s_{\alpha,\alpha}, t_{\beta,\beta'}, r_{\theta,\square}]$. For the two conditions, $[\cdots, s_{\alpha,\alpha}, t_{\beta,\beta'}, r_{\theta,\square}, \cdots]$ could be obtained in P_s(figure 10.a) or P_r(figure 10.b). So m_{k+1} could also be inserted into the Z-cycle.

case IV : For P_s, $send(m_{k+1}) \xrightarrow{hb} receive(m_k)$ and for P_t, $send(m_1) \xrightarrow{hb} send(m_{k+1})$. That is, P_s and P_t are *non-casual*.

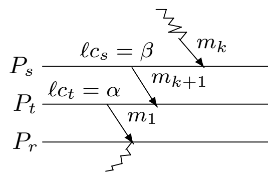

Figure 11.

For P_t, when P_t sends m_1 to P_r. Z_Queue_t contains $[t_{\square,\alpha}, r_{\square,\square}]$.When P_t receives m_{k+1}, Z_Queue_t contains $[s_{\square,\beta}, t_{\alpha,\square}]$. So after merge action, Z_Queue_t could contain $[s_{\square,\beta}, t_{\alpha,\alpha}, r_{\square,\square}]$ and then P_t requests P_r and merges again to obtain $[s_{\square,\beta}, t_{\alpha,\alpha}, r_{\gamma,\gamma'}, \cdots]$, for some γ'. For P_s, when P_s receives m_k, Z_Queue_s could have $[\cdots, s_{\beta,\square}]$. And Z_Queue_s already contains $[s_{\square,\beta}, t_{\square,\square}]$. So after merge action Z_Queue_s would contains $[\cdots, s_{\beta,\beta'}, t_{\square,\square}]$. Since there are two $\square$ symbols, P_s requests Z_Queue_t, which already contains $[s_{\square,\beta}, t_{\alpha,\alpha}, r_{\gamma,\gamma'}, \cdots]$, to merge again. So in Z_Queue_s there will be a Z-cycle $[\cdots, s_{\beta,\beta}, t_{\alpha,\alpha}, r_{\gamma,\gamma'}, \cdots]$. Hence m_{k+1} could also be inserted into the Z-cycle.

From above discussion of four distinct cases, the theorem still holds when the length of Z-cycle is $k+1$. By induction the proof is completed. ⊠

6. Conclusions

The task of detecting Z-cycles has never been implemented before, so there is not any evaluation of such a scheme as this. In this paper, we innovate an appropriate data structure expressing Z-path and detecting algorithm in distributed computing system. Although the algorithm demands much piggybacked Z-paths information, we can detect Z-cycles and involved checkpoints accurately. By Netzer,Xu's theorem and this algorithm we can distinguish useless checkpoints(involved in a Z-cycle) from other checkpoints. Hence the objective of breaking Z-cycles could be accessible by inserting minimal number of forced checkpoints. In the future, we can eliminate useless checkpoints or rearrange their position to make Z-cycle free for decreasing the number of forced checkpoints to destroy Z-cycles is still an important issue.

7. References

[1] L.Lamport. Time, Clocks and the Ordering of Evevts in a Distributed System. *Comm. ACM,* vol.21, no.7, pp.558-565, 1978

[2] R.H.B Netzer and J. Xu. Necessary and Sufficient Conditions for Consistent Global Snapshots. *IEEE Trans. on Parallel and Distributed Systems*, vol.6, no.2, pp.165-169, 1995

[3] Taesoon Park. Heon Y. Yeon. Application Controlled Checkpointing Coordination for Fault-Tolerant Distributed Computing Systems. Dept of Computer Engineer Sejong University. *Parallel Computing*, vol.26, no.4, pp.467-482, 2000

[4] D. Manivannan and M. Singhal. Quasi-Synchronous Checkpointing: Models, Characterization, and Classification. *IEEE Trans. on Parallel and Distributed Systems*, vol.10, no.7, pp.703-713, 1999

[5] D. Briatico, A. Ciuffoletti and L. Simoncini, A distributed domino-effect free recovery algorithm. In *Proc. of the IEEE 4th Symp. on Reliability in Distributed Software and Database Systems*, pp. 207-215 , 1984

[6] J.M. Helary et al. Communication-based prevention of useless checkpoints in distributed computations. *Distributed Computing*, vol.13, no.1, pp.29-43, 2000

[7] R.Baldoni, F. Quaglia, and B. Ciciani. A *VP*-accordant checkpointing protocol preventing useless checkpoints. In *the 17th IEEE Symposium on Reliable Distributed Systems*, pp.61-67. 20-23 Oct. 1998

[8] Yi-Min Wang. Maximum and Minimum Consistent Global Checkpoints and their Applications. In *the 14th IEEE Symposium on Reliable Distributed Systems* , pp.86-95. 13-15 September, 1995

[9] Yi-Min Wang. Consistent Global Checkpoints that Contain a Given Set of Local Checkpoints.

IEEE Transactions on Computers, vol.46, no.4, pp.456-468, 1997.

[10] Jane-Feng Chiu and Ge-Ming Chiu. Placing Forced Checkpoints in Distributed Real-Time Embedded Systems. *IEEE Computing & Control Engineering Journal*, vol.13, issue 4, pp.197-205 Aug 2002

[11] B. Randell. System structures for software fault-tolerance. *IEEE Transactions on Software Eng.*,vol.1 no.2, pp.220-232, June, 1975

[12] R. Baldoni, J. M. Helary, and M. Raynal. Rollback-dependency trackability: Visible characterizations. In *18th ACM Symposium on the Principles of Distributed Computing(PODC'99)*, Atlanta(USA), pp.33-42, May 1999

[13] I. C. Garcia and L. E. Buzato. On the minimal characterization of rollback-dependency trackability property. In *Proceedings of the 21th IEEE Int. Conf. on Distributed Computing Systems*, pp.0342-0349. 16-19 April 2001

[14] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. on Computer Systems*,vol.3, no.1, pp.63-75, Feb, 1985

[15] R. Koo and S. Toueg. Checkpointing and Rollback-recovery for distributed systems. *IEEE Trans. on Software Eng.*, vol.13, no.1, pp.23-31, Jan, 1987

[16] R.D. Schlichting and F.B. Schneider. Fail-Stop Processors: an Approach to Designing Fault-Tolerant Computing Systems. *ACM Trans. on Computer Systems*, vol.1, no.3, pp.222-238, 1983

[17] E.N. Elnozahy, D.B. Johnson and Y.M. Wang. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Surveys(CSUR)*, vol.34, issue 3, pp.375-408, Sep. 2002

Appendix :

The section illustrate our algorithms detailed and we typeset them with one column.

Actions taken when P_i sends a message M to P_j

1: **for** each Z-path in Z_Queue_i **do**
2: Duplicate the front part $[\cdots\cdots, \underset{\alpha,\Box}{i}]$, where $\alpha \leq \ell c_i$ to merge $[\underset{\Box,\ell c_i}{i}, \underset{\Box,\Box}{j}]$ into a new Z-path $[\cdots\cdots, \underset{\alpha,\ell c_i}{i}, \underset{\Box,\Box}{j}]$ and copy them into $Z_Queue_buffer1_i$;
3: **end for**
4: Send ($Z_Queue_buffer1_i$ and M) to P_j;
5: Clear $Z_Queue_buffer1_i$; // end

Actions taken when P_i receives a message (M, $Z_Queue_buffer1_k$) from P_k

1: Store $Z_Queue_buffer1_k$ into $Z_Queue_buffer1_i$;
2: **for** each Z-path $[\cdots\cdots, \underset{\cdot,\alpha}{k}, \underset{\Box,\Box}{i}]$ in $Z_Queue_buffer1_i$ **do**
3: Write ℓc_i into it, $[\cdots\cdots, \underset{\cdot,\alpha}{k}, \underset{\ell c_i,\Box}{i}]$
4: **end for**
5: Update(csn_i , $Z_Queue_buffer1_i$);
6: PruneZ-path(csn_i , $Z_Queue_buffer1_i$);
7: **for** each $[\cdots, \underset{\cdots,\ell c_i}{i}, \underset{\Box,\Box}{j}]$ appears in Z-path of Z_Queue_i **do**
8: Send Z-path $request([\underset{\Box,\ell c_i}{i}, \underset{\Box,\Box}{j}])$ to P_j to obtain the back part $[\underset{\Box,\ell c_i}{i}, \underset{\cdots,\cdots}{j}, \cdots]$ of Z-paths in their Z_Queue_j;
9: Obtain Z-paths $[\underset{\Box,\ell c_i}{i}, \underset{\cdots,\cdots}{j}, \cdots]$ from other processe js and connect them with $[\cdots, \underset{\cdots,\ell c_i}{i}, \underset{\Box,\Box}{j}]$ into $[\cdots, \underset{\cdots,\ell c_i}{i}, \underset{\cdots,\cdots}{j}, \cdots]$;
10: PruneZ-path(csn_i , Z_Queue_i);
11: **end for**
12: **for** each $z-path$ in $Z_Queue_buffer1_i$ **do**
13: Take the front part $[\cdots, \underset{\cdots,\cdots}{k}, \underset{\alpha,\Box}{i}]$, where $\alpha = \ell c_i$
14: **for** each $z-path$ containing $[\cdots, \underset{\cdots,\ell c_i}{i}, \cdots]$ in Z_Queue_i **do**
15: Connect $[\cdots, \underset{\cdots,\cdots}{k}, \underset{\alpha,\Box}{i}]$ with $[\underset{\Box,\ell c_i}{i}, \cdots]$ and then generate a new Z-path $[\cdots, \underset{\cdots,\cdots}{k}, \underset{\alpha,\ell c_i}{i}, \cdots]$;
16: CheckZ-cycle(this new Z-path, $[\underset{\Box,\cdots}{k}, \underset{\ell c_i,\Box}{i}]$);
17: **end for**
18: **end for**
19: Clear $Z_Queue_buffer1_i$ and $Z_Queue_buffer2_i$;
20: Processing M ; // end

Actions taken when P_i takes a basic checkpoint

1: P_i takes a checkpoint $C_{i,\ell c_i}$;
2: $\ell c_i := \ell c_i + 1$;
3: $csn_i[i] := \ell c_i$;
4: PruneZ-path(csn_i , Z_Queue_i); // end

Actions taken when P_i receives a Z-path $request([\underset{\square,\ell c_q}{q}, \underset{\square,\square}{i}])$ **from** P_q

1: **for** each Z-path of Z_Queue_i **do**
2: Cut the back part $[\underset{\square,\ell c_q}{q}, \underset{\cdots,\cdots}{i}, \cdots\cdots]$ of the Z-path;
3: **if** the Z-path is as $[\underset{\square,\ell c_q}{q}, \underset{\cdots,\cdots}{i}, \cdots, \underset{\cdots,\alpha}{r}, \underset{\square,\square}{s}]$ **then**
4: Send Z-path $request([\underset{\square,\alpha}{r}, \underset{\square,\square}{s}])$ to all processes s to obtain back part $[\underset{\square,\alpha}{r}, \underset{\cdots,\cdots}{s}, \cdots\cdots]$ of z-paths in their Z_Queue and wait for reply;
5: Collect Z-paths $[\underset{\square,\alpha}{r}, \underset{\cdots,\cdots}{s}, \cdots\cdots]$ from other processes and connect with $[\underset{\square,\ell c_q}{q}, \underset{\cdots,\cdots}{i}, \cdots, \underset{\cdots,\alpha}{r}, \underset{\square,\square}{s}]$, then update this Z-path $[\cdots, \underset{\square,\ell c_q}{q}, \underset{\cdots,\cdots}{i}, \cdots, \underset{\cdots,\alpha}{r}, \underset{\cdots,\cdots}{s}, \cdots]$;
6: Store $[\underset{\square,\ell c_q}{q}, \underset{\cdots,\cdots}{i}, \cdots, \underset{\cdots,\alpha}{r}, \underset{\cdots,\cdots}{s}, \cdots]$ into $Z_Queue_buffer1_i$
7: **else**
8: Store the Z-path $[\underset{\square,\ell c_q}{q}, \underset{\cdots,\cdots}{i}, \cdots\cdots]$ into $Z_Queue_buffer1_i$;
9: **end if**
10: **end for**
11: Send $Z_Queue_buffer1_i$ back to P_q for reply.
12: Update(csn_i;$Z_Queue_buffer1_i$);
13: PruneZ-path(csn_i , Z_Queue_i);
14: Clear $Z_Queue_buffer1_i$, $Z_Queue_buffer2_i$; // end

Procedure Update(csn , Z_Queue)

1: **for** each $z - path$ in Z_Queue **do**
2: **for** each $Pid.\ell c_out$ **do**
3: $csn[Pid] = max(csn[Pid], Pid.\ell c_out - 1)$;
4: **end for**
5: **end for** // end

Procedure PruneZ-path(csn , Z_Queue)

1: **for** each Z-path in Z_Queue **do**
2: **while** first $Pid.\ell c_out \leq csn[Pid]$ **do**
3: Delete the first element of the z-path; // The sending of the first message occurred at the left side of checkpoint line ,so first element(message) is useless.
4: **end while**
5: **end for** // end

Procedure CheckZ-cycle($z - path$, $[\underset{\square,\alpha}{k}, \underset{\beta,\square}{i}]$)

1: **if** there exists m in $z - path$ $[\cdots, \underset{\cdots,out}{m}, \cdots, \underset{\cdots,\alpha}{k}, \underset{\beta,\cdots}{i}, \cdots, \underset{in,\cdots}{m}, \cdots]$ such that $in \leq out$ **then**
2: **if** there exists at least one Pid such that $\ell c_in < \ell c_out$ in the cycle $[\underset{\square,out}{m}, \cdots, \underset{\cdots,\alpha}{k}, \underset{\beta,\cdots}{i}, \cdots, \underset{in,\square}{m}]$ **then**
3: Z-cycle $[\underset{\square,out}{m}, \cdots, \underset{\cdots,\alpha}{k}, \underset{\beta,\cdots}{i}, \cdots, \underset{in,\square}{m}]$ forms and save it;
4: **end if**
5: **end if** // end