An Efficient Computing-Checkpoint Based Coordinated Checkpoint Algorithm

Men Chaoguang[1,2] Wang Dongsheng[1,2] Zhao Yunlong[2]

[1] Department of Computer Science and Technology, Tsinghua University, Beijing 100084, P.R.China
{mencg,wds}@tsinghua.edu.cn
[2] Research Center of High Dependability Computing Technology, Harbin Engineering University, Harbin, Heilongjiang, 150001, P.R.China

Abstract. In this paper, the concept of "computing checkpoint" is introduced, and then an efficient coordinated checkpoint algorithm is proposed. The algorithm combines the two approaches of reducing the overhead associated with coordinated checkpointing, which one is to minimize the processes which take checkpoints and the other is to make the checkpointing process non-blocking. Through piggybacking the information including which processes have taken new checkpoint in the broadcast committing message, the checkpoint sequence number of every process can be kept consistent in all processes, so that the unnecessary checkpoints and orphan messages can be avoided in the future running. Evaluation result shows that the number of redundant computing checkpoints is less than 1/10 of the number of tentative checkpoints. Analyses and experiments show that the overhead of our algorithm is lower than that of other coordinated checkpoint algorithms.

1 Introduction

Checkpointing and rollback-recovery has been an attractive technique for providing fault-tolerance in distributed computing system. When a fault occurs, the processes can reload the checkpoints state to recover the system [1-2]. Due to its simple, domino-free, and the minimal requirement for storage, coordinated checkpointing is efficient. Two approaches are used to reduce the overhead of coordinated checkpoint algorithm: one is to minimize the number of checkpoints [3-6]; the other is to make the checkpointing process non-blocking [7-8]. To reduce the overhead of coordinated checkpoint algorithm, the concept of computing checkpoint is introduced. And, an efficient coordinated checkpoint algorithm based on computing checkpoint is proposed.

The paper is organized as follows: Section 2 introduces the system model and definitions. Section 3 presents an efficient low-cost non-blocking coordinated checkpoint algorithm (*LNCC*). Section 4 gives its correctness proofs. Section 5 evaluates the number of redundant computing checkpoints. Section 6 shows the experiment results. Section 7 compares *LNCC* with some earlier relative coordinated checkpoint schemes. Section 8 draws a conclusion.

2 Preliminaries

The distributed computation consists of N sequential processes denoted by $P_1, P_2, \ldots, P_N$ running concurrently on fail-stop. Processes do not share a common memory or a common clock. Message passing is the only way for the processes to communicate with each other. The computation is asynchronous, i.e., each process progresses at its own speed and messages are exchanged through reliable communication channels, whose transmission delays are finite but arbitrary. The messages generated by the underlying distributed application will be referred to as computation messages. The messages generated by the processes to advance checkpoints will be referred to as system messages. A process can execute internal, send and delivery statements. Each process P_i produces a sequence of events $e_{i,1}, \ldots, e_{i,s}, \ldots$, which can be finite or infinite. Every process P_i has an initial local state denoted $\sigma_{i,0}$. The sequence of events $e_{i,1}, \ldots, e_{i,s}$ applied to the initial state $\sigma_{i,0}$ result the state $\sigma_{i,s}$. Every process saves its local state on stable memory to produce its local checkpoint and each checkpoint taken by a process is assigned a unique checkpoint sequence number (*CSN*). The checkpoint taken by the initiator or a process on which the initiator depends is called basic checkpoint. The i^{th} ($i \geq 0$) checkpoint of process P_k is assigned a sequence number i and is denoted by $C_{k,i}$. Any event $e_{k,x}$ exist between $C_{k,i-1}$ and $C_{k,i}$ is said "$e_{k,x}$ belongs to $C_{k,i}$". The i^{th} checkpoint interval of process P_p denotes all the computation performed between its i^{th} and $(i+1)^{th}$ checkpoint, including the i^{th} checkpoint but not the $(i+1)^{th}$ checkpoint, denoted as $I(p,i)$. In distributed systems, orphan messages and in-transit messages may result in the inconsistency.

Orphan messages: A message M sent by process P_i to process P_j is called an orphan message with respect to the ordered pair of local checkpoints ($C_{i,x}$,$C_{j,y}$), if the delivery of M belongs to $C_{j,y}$ while its sending event does not belong to $C_{i,x}$.

In-transit messages: A message M sent by process P_i to process P_j is called an in-transit message with respect to the ordered pair of local checkpoints ($C_{i,x}$,$C_{j,y}$), if the sending of M belongs to $C_{i,x}$ while its delivery does not belong to $C_{j,y}$.

If a fault occurs, in-transit messages will be lost. By logging and replaying them out when process recovering, the in-transit messages lost can be avoided. An orphan message will result in the system becoming inconsistent when rollback recovery.

Definition 1, dependency relation: A process P_i sends a message to process P_j with respect to the ordered pair of local checkpoints ($C_{i,x}$,$C_{j,y}$), we say that P_j at its y^{th} checkpoint interval depends on P_i at its x^{th} checkpoint. Simply we say P_j depends on P_i, denoted as $R_j(i)$=1. If P_j depends on P_k, and P_k depends on P_i, we say P_j transitively depends on P_i. We simply call the two cases P_j depends on P_i.

Definition 2, computing checkpoint: Assume that P_i has taken its $(x+1)^{\text{th}}$ tentative checkpoint and sends a computation message M to P_j. Before receiving M, P_j knows P_i in its x^{th} checkpoint. Hence P_j must take forced checkpoint before delivering M. The checkpoint taken by P_j is called a computing checkpoint.

Definition 3, global consistent checkpoint: A global checkpoint is a set of local checkpoints, one from each process. A global checkpoint is consistent if no message is orphan with respect to any pair of its local checkpoints [9-10].

3 The Low-Cost Non-blocking Coordinated Checkpointing (*LNCC*)

Two-phase scheme and computing checkpoint are used to improve the efficiency of algorithm. When a process takes a computing checkpoint, it does not request these processes on which it depends to take checkpoints. A computing checkpoint should be transformed to a tentative checkpoint or be discarded according to the process receiving request or not. In the second phase, the initiator broadcasts committing message to all processes in the system, piggybacking the information including which processes have taken checkpoints. According to the information, each process can ensure the *CSNs* of all processes are consistent so that orphan message and unnecessary checkpoint can be avoided. When the checkpoints are taken, the dependent relations of the processes will be updated to avoid taking unnecessary checkpoints [11].

3.1 The Data Structure of *LNCC*

R_i: a Boolean array. $R_i(j)$=1 means P_i depends on P_j. R_i is initialized to 0, but $R_i(i)$=1.
Tem_R_i: a Boolean array. It is used to save temporary dependent relation when taking tentative checkpoint. It is initialized to 0, but $Tem\_R_i(i)$=1 in every P_i.
Rep_R_i: a Boolean array. It is used to save which process has taken a new checkpoint.
$CSN_i[j]$: an integer array. $CSN_i[j]=X$ means process P_j takes *Xth* checkpoint that P_i expects. *CSN* is initialized to 0 in every process.
Cp_state: a Boolean variable. Cp_state_i=1 marks a process in its checkpointing.
Com_state: a Boolean variable, marking a process takes a computing checkpoint.
Weight: a non-negative variable of type real with maximum value of 1. It is used to detect the termination of the checkpointing.
Trigger: a tuple (*pid,inum*). *pid* indicates the checkpoint initiator that triggered this node to take its latest basic checkpoint. *inum* indicates the *CSN* at node *pid* when it takes its local basic checkpoint on initiating consistent checkpointing.

3.2 The *LNCC* Algorithm Description

A formal description of the two-phase checkpoint algorithm is given in Fig.1.

3.3 An Example of *LNCC* Algorithm

Fig.2 is an example of *LNCC* executing. Solid line means transmitting computing message and dashed line means request message. P_4, as the initiator, takes checkpoint $C_{4,1}$ and sends request to the processes on which it depends. After taking checkpoint $C_{3,1}$, P_3 sends *M4* to P_2 with $CSN_3(3)$=1. Due to $CSN_2(3)$=0 and $CSN_2(3)<CSN_3(3)$, P_2 takes computing checkpoint $C_{2,1}$ before delivering *M4*. Due to $CSN_1(2)$=0 and $CSN_2(2)$=1, P_1 takes computing checkpoint $C_{1,1}$ before delivering *M5*. After receiving request, P_1 makes computing checkpoint $C_{1,1}$ tentative and sends request to P_2. P_2 makes computing checkpoint $C_{2,1}$ tentative. The system is consistent. When receiving

committing message, P_6 cancels the dependent relation of P_6 depending P_5. P_6 increases $CSN_6(5)$, $CSN_6(4)$, $CSN_6(3)$, $CSN_6(2)$, $CSN_6(1)$.

Actions taken when P_i sends a computation message to P_j :

If P_i is in its checkpointing, P_i sends message with its *CSN* and *Trigger*.

Actions for the initiator P_j :

The initiator P_j increases its $CSN_j[j]$, sets *weight*:=1, *trigger*:=(P_j, $CSN_j[j]$), marks that it is in its checkpointing and takes local checkpoint. The initiator sends request message with a half of its residuary *weight* to the processes on which the initiator depends to request them take checkpoints too.

Actions at process P_i , on receiving a checkpoint request from P_j :

If P_i has taken a computing checkpoint, it makes computing checkpoint basic tentative, propagates the checkpoint request to these processes with a half of its residuary *weight* on which it depends but P_j does not depend. P_i replies message with residuary *weight* to the initiator. If P_i doesn't take computing checkpoint and $CSN_j[j]>CSN_i[j]$, P_i increases $CSN_i[i]$, takes tentative checkpoint, propagates the checkpointing requiring with a half of its residuary *weight* to the processes on which it depends but P_j does not depend. P_i replies message with residuary *weight* to the initiator.

Actions at process P_i, on receiving a computation message from P_j :

P_i receives a computation message from P_j with $CSN_j[j]$ and *trigger*. If $CSN_j[j]>CSN_i[j]$ then P_i takes a computing checkpoint, increases $CSN_i[i]$, then delivers the message.

Actions in the second phase for the initiator P_i:

If the sum of *weight* which piggyback in every reply messages is equal to one, it means all processes on which initiator depends have taken checkpoints, the initiator broadcasts committing message with the information including which processes have taken checkpoint; otherwise initiator broadcast negative message. The initiator updates the dependent relations.

Actions at other process P_j on receiving a broadcast message from P_i:

If a process receives a committing message, the receiver makes tentative permanent or discards computing checkpoint. The receiver updates the dependent relations and the *CSN*s of each process according to the information that which processes have taken checkpoints. If a process receives a negative message, the receiver discards tentative checkpoint or computing checkpoint, and updates the dependent relations and *CSN*s of each process.

Fig.1. The *LNCC* algorithm description

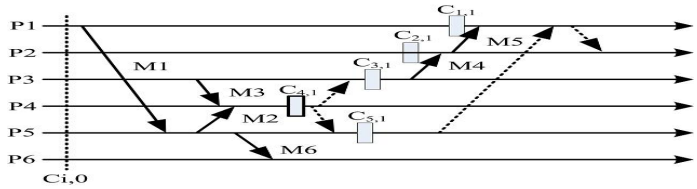

Fig.2. An example of a distributed system with *LNCC* algorithm

4 Correctness of the algorithm

Theorem 1: Computing checkpoint is necessary.
Proof: Assume that P_j sends M piggybacking $CSN_j[j]$ to P_i. If $CSN_j[j]>CSN_i[j]$, it means P_j has taken a checkpoint before sending M. Assume that P_i doesn't take computing checkpoint before delivering M. Since the future running situations of processes are unforeseen, later, P_i may receive a request from another process P_k. P_i will take checkpoint, M becomes an orphan. If P_i takes computing checkpoint before delivering M, when receiving a checkpoint request, P_i will transform the computing checkpoint to basic tentative checkpoint, so M is avoided to become an orphan. After making computing checkpoint tentative, P_i propagates request of taking checkpoint to the processes on which it depends but P_k does not depend. If P_i doesn't receive any request message, the computing checkpoint will be discarded, and the system still is consistent.□
Theorem 2: An initiator P_i takes checkpointing, all the processes on which P_i depends should take relative checkpoints too.
Proof: If initiator P_i directly depends on a process P_j, there is $R_i(j)=1$. P_j will receive a request from P_i. So P_j will take tentative checkpoint caused by P_i. If the initiator P_i transitively depends on P_j, there must be processes $P_1, P_2, \ldots, P_n$, having P_i directly depends on P_1, P_1 directly depends on $P_2, \ldots, P_n$ directly depends on P_j. P_j will receive the request and take tentative checkpoint.□
Theorem 3: *LNCC* is a consistent checkpoint algorithm.
Proof: Assume that there is an inconsistent after the *LNCC*. There is a message M sent from P_i to P_j such that P_j saves the event of delivering M and P_i doesn't save the event of sending M. P_j is an initiator or a process on which initiator depends because of its taking a checkpoint. M is sent from P_i to P_j, so there is $R_j(i)=1$. If P_j takes checkpoint, P_i must take checkpoint too. Contradiction.□

5 Evaluating the redundant computing checkpoints

A computing checkpoint that isn't transformed into a tentative is a redundant checkpoint. If there is not any redundant computing checkpoint, the checkpoint algorithm is a minimum algorithm. We analyze the proportion of redundant computing checkpoint among all checkpoints.

5.1 The model and assumption

A checkpoint interval can be denoted by two parts: the period of not taking checkpointing (denoted as T_{NC}) and the period of taking checkpointing (denoted as T_C) as shown in Fig.3.
Obviously there is $T_{NC}>>T_C$. P_{sc} initiates checkpointing and the processes on which P_{sc} depends take checkpoint too. The set N_D includes the processes on that P_{sc} depends and the set $\overline{N_D}$ includes the processes on that P_{sc} does not depend. Computing

checkpoints are produced in period of T_C only. The computing checkpoints that are produced in N_D will be transformed into tentative and the computing checkpoints that are produced in $\overline{N_D}$ are redundant computing checkpoints. In order to compute the redundant checkpoint, denoted as N_{comp}, assume that the message sending and receiving rate are the same, denoted as λ_M, and receiver receives message in no delay.

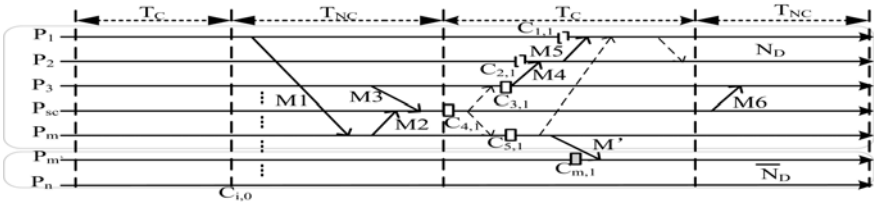

Fig.3. The example of redundant computing checkpoints

5.2 The number of processes on that initiator does not depend

In the last T_{NC} period, the number of messages received by P_{sc}, $N_{SC}(T_{NC})$, is:

$$P\{N_{SC}(T_{NC})=m\}=\frac{(\lambda_M T_{NC})^m}{m!}e^{\lambda_M T_{NC}} \tag{1}$$

The expectation number, N_{SC}, is:

$$\begin{aligned} N_{SC} &= E\big(N_{SC}(T_{NC})\big)=\sum_{m=0}^{\infty} m\frac{(\lambda_M T_{NC})^m}{m!}e^{\lambda_M T_{NC}} \\ &= \lambda_M T_{NC} e^{-\lambda_M T_{NC}} \times e^{\lambda_M T_{NC}} = \lambda_M T_{NC} \end{aligned} \tag{2}$$

The probability that P_{sc} receives a message from P_i is $\rho = 1/(N-1)$. The probability that P_{sc} does not receive a message from P_i is:

$$\rho(0)=(1-\rho)^{N_{SC}}=(1-1/(N-1))^{\lambda_M T_{NC}} \tag{3}$$

The probability that K out of $(N-1)$ processes does not send messages to P_{sc} is:

$$P(x=K)=\binom{N-1}{K}\rho(0)^K(1-\rho(0))^{N-1-K} \tag{4}$$

Its expectation number is:

$$\begin{aligned} N_{ND} = E(K) &= \sum_{K=0}^{K=N-1} KP(K)=\sum K\binom{N-1}{K}\rho(0)^k(1-\rho(0))^{N-1-K} \\ &= (N-1)\rho(0)=(N-1)(1-1/(N-1))^{\lambda_M T_{NC}} \end{aligned} \tag{5}$$

That is, in the last T_{NC} period, the expectation number of processes that have not sent any message to P_{sc} equals to N_{ND}. Assume that there are two sets, S_{SI}={P_i|P_i has sent a message to P_{sc} in the last T_{NC} period} and S_{NSI}={P_i|P_i has not sent any message to P_{sc}

in the last T_{NC} period}. Assume that $P_i \in S_{NS1}$ in the last T_{NC} period, the probability that the message sent by P_i has not sent to S_{S1} is:

$$\rho_{ND} = N_{ND} / N \quad . \tag{6}$$

In the last T_{NC} period, the expectation number of processes that belong to set S_{NS1} and have not sent any message to set S_{S1} is N_{ND1}.

$$N_{ND1} = N_{ND}\rho = N_{ND}(\rho_{ND})^{\lambda_M T_{NC}}$$

$$= (N-1)(1-1/(N-1))^{\lambda_M T_{NC}}((N-1)(1-1/(N-1))/N)^{\lambda_M T_{NC}} . \tag{7}$$

In turn, the processes which belong to S_{NS1} can be parted into two sets, S_{S2}={P_i|P_i has sent a message to S_{S1} in the last T_{NC} period} and S_{NS2}={P_i|P_i has not sent any message to S_{S1} in the last T_{NC} period}. $|S_{NS2}| = N_{ND1}$ .The number of processes which belong to S_{NS2} and not send message to S_{S1} is N_{ND2}.

$$N_{ND2} = N_{ND1}\rho_2 \quad . \tag{8}$$

Thereinto,

$$\rho_2 = (\rho_{ND2})^{\lambda_M T_{NC}} \quad . \tag{9}$$

$$\rho_{ND2} = N_{ND1} / N \quad . \tag{10}$$

In turn, the processes which belong to $S_{NS(i-1)}$ can be parted into two set, S_{Si}={P_i|P_i has sent a message to $S_{S(i-1)}$ in the last T_{NC} period} and S_{NSi}={P_i|P_i has not sent any message to $S_{S(i-1)}$ in the last T_{NC} period}. The probability of a process P_i which belongs to S_{NSi} and does not send message to $S_{S(i-1)}$ is:

$$\rho_{NDi} = N_{ND(i-1)} / N \quad . \tag{11}$$

The probability of the processes which belong to S_{NSi} and don't send message to $S_{S(i-1)}$ is:

$$\rho_i = (\rho_{NDi})^{\lambda_M T_{NC}} \quad . \tag{12}$$

The expectation number of processes which belong to S_{NSi} and don't send message to $S_{S(i-1)}$ is:

$$N_{ND(i+1)} = N_{NDi}\rho_i \quad . \tag{13}$$

If $(N_{NDi}-N_{ND(i+1)}) \leq 1$, the number of processes on which P_{SC} does not depends is $N_{ND(i+1)}$. Set $N_{ID} = N_{ND(i+1)}$, N_{ID} is the number of processes on which P_{SC} does not depend.

5.3 The number of redundant computing checkpoints

Assume that there are two sets, S_{ID}={P_i|P_{SC} does not depend on P_i directly or indirectly} and S_D={P_i|P_{SC} depends on P_i directly or indirectly}.

In the T_C period, P_i that belongs to S_{ID} receives a message sent by a process P_j that belongs to S_D. If the *CSN* which is appended by P_j is larger than the *CSN* which P_i expects P_j has, P_i must take computing checkpoint. This computing checkpoint is redundant computing checkpoint that should be discarded in the future. If the *CSN* which is appended by P_j isn't larger than the *CSN* which P_i expects P_j has, P_i does not take computing checkpoint. For simplifying analysis, we consider the worst situation that P_i takes computing checkpoint when it receiving a message sent from P_j, regardless its *CSN* and the appended *CSN*.

The ratio that the processes belonged to S_{ID} receive the messages sent from the processes belonged to S_D is λ_D.

$$\lambda_D = ((N - N_{ID})/N)\lambda_M \quad . \tag{14}$$

In the T_C period, the probability of taking computing checkpoint is:

$$\rho_C = 1 - e^{-\lambda_D T_C} \quad . \tag{15}$$

The expectation number of redundant computing checkpoints is:

$$N_{comp} = E(K_C) = \sum_{K=0}^{N_{ID}} K \binom{N_{ID}}{K} \rho_C{}^K (1-\rho_C)^{N_{ID}-K}$$

$$= N_{ID}\rho_C = N_{ID}(1 - e^{-((N-N_{ID})/N)\lambda_M T_C}) \quad . \tag{16}$$

Table 1 shows the redundant computing checkpoints ratio（$E\%$） to tentative checkpoints under N=20.

Table 1. The efficiency of *LNCC* algorithm under parameters

T_{NC}	300	300	600	600	300	300	600	600
T_C	10	10	10	10	20	20	20	20
λ_M	0.01	0.001	0.01	0.001	0.01	0.001	0.01	0.001
N_{ID}	11.97	18.12	7.29	17.27	11.97	18.12	7.29	17.27
ρ_C	0.04	0.00046	0.061	0.0014	0.07	0.0019	0.12	0.0027
N_{comp}	0.48	0.08	0.45	0.23	0.84	0.034	0.86	0.047
E (%)	5.6	4	3.4	7.8	9.5	1.8	6.3	1.7

Table 1 shows that the number of redundant computing checkpoints depends on the values of T_{NC}, T_C and λ_M, but it is always less than 10% of the total tentative checkpoints. The computing checkpoints ratio is in the worst situation. In fact the real ratio is less than the ratio listed in table 1, because the action of taking computing checkpoint relies on the *CSNs* of processes too.

6 Experiment

A system with 16 nodes is simulated, the nodes are connected through a LAN which has 100Mbps bandwidth and each node has one process running on it. The length of each computation message is 1KB, and the length of each system message is 50Bytes. The computer's CPU is PⅣ2.4GHz, memory is 512MB. The rate of memory bus with 64bit width is 100MHz. The length of checkpoint is 1MB. Fig.4 shows the number of redundant computing checkpoints and tentative checkpoints change with the change of message sending rate.

As shown in Fig.4, when the massage sending rate increasing, the number of redundant computing checkpoints increases at first, then gets its maximum, and then decreases. This can be explained as follows: a process takes a computing checkpoint only when it receives a computation message from a process that has taken a tentative checkpoint. If the message sending rate is low, processes have low probability of sending messages and they have low probability of receiving messages during the T_C

time. Thus, processes have low probability of taking computing checkpoints. So, the number of redundant computing checkpoints is less too. If the message sending rate enhancing, it is more likely for a process to receive a message and take a computing checkpoint during the T_C time. If the message sending rate enhancing further, the initiator is more likely to depend on other processes. The computing checkpoints are also more likely to be turned into tentative checkpoint. The number of redundant computing checkpoint decreases. If the message sending rate is large enough, the number of redundant computing checkpoint will be zero. Simulations show that the number of redundant computing checkpoints always less than 5 percent of the tentative checkpoints, less than the number computed in table 1. Because we assume that a process always takes a computing checkpoint when the process receives a computation message from another process that has taken tentative checkpoint in table 1, in despite of the *CSN* of them. Fig.5 shows the comparison result of the algorithms' overhead. As shown in Fig.5, *LNCC* has the low overhead than other algorithms.

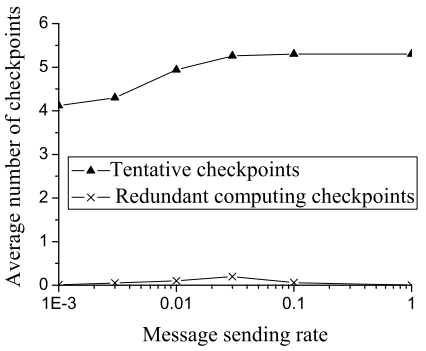

Fig.4. The comparison of redundant computing checkpoints and tentative checkpoints

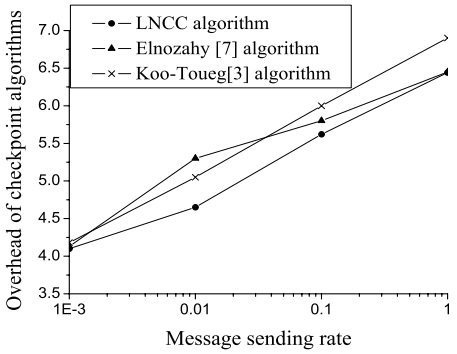

Fig.5. The comparison of the overhead of checkpoint algorithms

7 Comparisons with existing work

Many coordinated checkpointing schemes have been proposed for the distributed computing. In ref.[3], a min-process coordinated checkpointing scheme has been proposed, in which only minimal processes need to take new consistent globe checkpoint while others needn't change their old checkpoints. But it must block processes when taking checkpointing. Blocking algorithms may dramatically degrade system performance [7]. To address this issue, non-blocking algorithms are proposed [7-8]. In the algorithms, processes use a checkpoint sequence number to identify orphan messages. However, these algorithms require all processes to take checkpoints during checkpointing, even though many of them may not be necessary.

Prakash–Singhal's algorithm was the first algorithm to attempt to combine these two approaches [12]. It only forces minimum number of processes to take checkpoints and does not block the underlying computation during the checkpointing. However, this algorithm may result in an inconsistency [6,11]. Cao-Singhal improves Prakash–Singhal's algorithm by using mutable checkpoint [11]. According to Cao-Singhal's

algorithm, when P_i receives a computation message M from P_j, P_i should take a mutable checkpoint if following three conditions have been satisfied: (1) P_j is in checkpointing process before sending M; (2) P_i has sent a message since last checkpoint; (3) P_i has not taken a checkpoint associated with the initiator [11]. But there is still an inconsistency in some situations.

Fig.6 illustrates the inconsistency of Cao-Singhal's algorithm. P_4 (as an initiator) takes checkpoint and asks P_3 and P_5 to take checkpoints (dashed represents request messages). After taking a checkpoint, P_3 sends M_4 to P_2, and P_2 sends M_5 to P_1. Condition 2 isn't satisfied, so P_2 needn't take mutable checkpoint. Due to transmission delays, later, P_1 may receive a checkpoint request after receives M_5. Then P_1 takes a checkpoint and requires P_2 to take checkpoint. Then M_4 becomes an orphan.

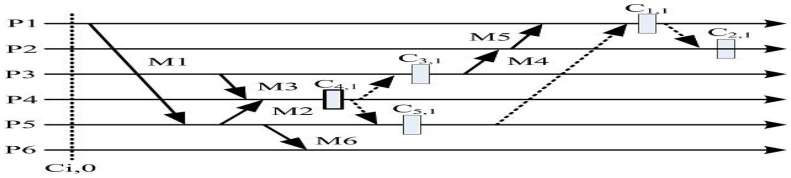

Fig.6. An inconsistent example of Cao-Singhal's algorithm

In table 2, we use four parameters to compare *LNCC* with other algorithms: the number of tentative checkpoints required during a checkpointing process, the blocking time (in the worst case), the system message overhead, whether the algorithm is distributed or not.

Table 2. A comparison of system performance

algorithm	checkpoints	blocking time	messages	distributed
Koo–Toueg[3]	N_{min}	$N_{min}*T_{ch}$	$3*N_{min}*N_{dep}*C_{uni}$	yes
Cao-Singhal [6]	N_{min}	$2*T_{msg}$	$C_{broad}+2*C_{uni}*(N+N_{min})$	yes
Elnozahy [7]	N	0	$2*C_{broad}+N*C_{uni}$	no
LNCC	$N_{min}+N_{comp}$	0	$\approx 2*C_{uni}*N_{min}+C_{broad}$	yes

C_{uni}: cost of sending a message from one process to another process; C_{broad}: cost of broadcasting a message to all processes; T_{disk}: delay incurred in saving a checkpoint on the stable storage; T_{data}: delay incurred in transferring a checkpoint to the stable storage; T_{msg}: delay incurred by transferring system messages during a checkpointing process; T_{ch}: the checkpointing time, $T_{ch}=T_{msg}+T_{data}+T_{disk}$; T_{comp}: delay incurred in saving a computing checkpoint; N_{min} is the number of processes that need to take checkpoints using the Koo–Toueg algorithm [3], N is the total number of processes in the system, N_{comp} is the number of redundant computing checkpoints during a checkpointing, N_{dep} is the average number of processes on which a process depends.

Since a computing checkpoint can be saved on the main memory, the delay of saving computing checkpoint can be ignored comparing with the delay of saving tentative checkpoint. The overhead of *LNCC* is $N_{min}*T_{disk}+N_{comp}*T_{comp}+2*C_{uni}*N_{min}+C_{broad}$, which is less than the overhead of other algorithms.

8. Conclusion

In this paper, a low-cost non-blocking coordinated checkpoint algorithm is presented. Through using computing checkpoint and piggybacking the information including which processes have taken checkpoint in the broadcast committing message, the unnecessary checkpoints and orphan messages can be avoided in the future running. The algorithm is consistent coordinated checkpoint algorithm which combines the two approaches of reducing the overhead associated with coordinated checkpointing. Analyses and simulations show that our algorithm is better than other coordinated checkpoint algorithms.

References

1. E.N.Elnozahy, L.Alvisi, Y.M.Wang and D.B.Johnson: A Survey of Rollback-Recovery Protocols in Message-Passing Systems. ACM Computing Surveys. 2002, 34(3):375-408.
2. S.Kalaiselvi, V.Rajaramana: A Survey of Checkpointing Algorithms for Parallel and Distributed Computers. Sadhana Academy Proceedings in Engineering Sciences. 2000, 25(5):489-510.
3. R.Koo, S.Toueg: Checkpointing and Rollback-Recovery for Distributed Systems. IEEE Transactions on Software Engineering. 1987:13:23–31.
4. J.L.Kim, T.Park. An Efficient Protocol for Checkpointing Recovery in Distributed Systems. IEEE Transactions on Parallel and Distributed Systems. 1993, 5(8):955–960.
5. Y.Deng, E.K.Park: Checkpointing and Rollback-Recovery Algorithms in Distributed Systems. Journal of Systems Software. 1994, 4:59–71.
6. Cao Guohong, M.Singhal: On the Impossibility of Min-Process Non-Blocking Checkpointing and an Efficient Checkpointing Algorithm for Mobile Computing Systems. Proceedings of the 27th int'l International Conference on Parallel Processing, Minneapolis, USA. 1998:37-44.
7. E.N.Elnozahy, D.B.Johnson, W.Zwaenepoel: The Performance of Consistent Checkpointing. Proceedings of the 11th Symposium on Reliable Distributed Systems, Houston. 1992:39-47.
8. L.M.Silva, J.G.Silva: Global Checkpointing for Distributed Programs. Proceedings of the 11th Symposium on Reliable Distributed Systems, Houston. 1992:155–162.
9. J.M.Helery, A.Mostefaoui, M.Raynal: Communication-Induced Determination of Consistent Snapshots. IEEE Transactions on Parallel and Distributed Systems. 1999, 10(9):865-877.
10. J.M.Helary, A.Mostefaoui, R.H.B.Netzer and M.Raynal: Preventing Useless Checkpoints in Distributed Computations. Proceedings of the 16th Symposium on Reliable Distributed Systems. 1997:183-190.
11. Cao Guohong, M.Singhal: Checkpointing with Mutable Checkpoints. Theoretical Computer Science. 2003,290:1127–1148.
12. R.Prakash, M.Singhal: Low-Cost Checkpointing and Failure Recovery in Mobile Computing Systems. IEEE Transactions on Parallel Distributed System. 1996, 7(10):1035–1048.