A Checkpointing Algorithm Based Unreliable Non-FIFO Channels

Chuanqing Shi\textsuperscript{a}, Shengfa Gao\textsuperscript{b}

College of Computer Science and Technology, ShanDong University, Jinan Shandong

Abstract

We propose a coordinated checkpointing algorithm based unreliable non-FIFO channel. In unreliable non-FIFO channel, the system can lose, duplicate, or reorder messages. The processes may not compute some messages because of message losses; the processes may compute some messages twice or more because of message duplicate; the processes may not compute messages according to their sending order because of message reordering. The above-mentioned problems make processes produce incorrect computation result, consequently, prevent processes from taking consistent global checkpoints. Our algorithm assigns each message a sequence number in order to resolve above-mentioned problems. During the establishing of the checkpoint, the consistency of checkpoint can be determined by the sequence number of sending and receiving messages. We can identify the lost messages, reordering messages and duplicate messages by checking the sequence number of sending and receiving messages. We resolve above-mentioned problems by resending the lost messages, buffering the reordering messages and dropping the duplicate messages. Our algorithm makes processes take consistent global checkpoints.

Index Terms: unreliable non-FIFO channel; message losses; message duplicate; message reordering; consistent global checkpoints

© 2012 Published by MECS Publisher. Selection and/or peer review under responsibility of the Research Association of Modern Education and Computer Science.

1. Introduction

Checkpointing and rollback-recovery are popular techniques that permit processes to make progress despite a process fails. We assume that the failures are transient problems. The failures are improbable to recur when the process restarts. With this scheme, a process takes a checkpoint periodically by saving its state on stable storage [1]. When a process has a failure, it rolls back to its most recent checkpoint that saves the state of this process and restarts execution.

Most checkpointing algorithms generally assume the communication channels are reliable FIFO channels [2, 3, 4]. Now, we propose a coordinated checkpointing algorithm based unreliable non-FIFO channel. In

The research in this paper is based on project supported by the subject of Natural Science Foundation of Shandong Province of China under grant Nos. Z2008G03.

* Corresponding author.
E-mail address: 'shichuanqing123@163.com; gsfyx@sdu.edu.cn
unreliable non-FIFO channel, the system can lose, duplicate, or reorder messages [5]. The processes may not compute some messages because of message losses; the processes may compute some messages twice or more because of message duplicate; the processes may not compute messages according to their sending order because of message reordering. The above-mentioned problems make processes produce incorrect computation result, consequently, prevent processes from taking consistent global checkpoints. Our algorithm can resolve these problems, and our algorithm can prevent “domino effect” and live problems associated with rollback-recovery.

The rest of the paper is organized as follows. Section 2 develops the necessary background. In Section 3, we describe a checkpointing algorithm based unreliable non-FIFO channels. The correctness proof is provided in Section 4. Section 5 concludes the paper.

2. Background

2.1. System Model

The distributed system considered in this paper consists of N+1 processes denoted by P_1, P_2, …, P_n, P_c. The processes denoted by P_1, P_2, …, P_n are ordinary processes and the process P_c is the coordinate process. The processes do not share a common memory or a common clock. Message passing is the only way for processes to communicate with each other. Each ordinary process progresses at its own speed and messages are exchanged through unreliable non-FIFO communication channel. P_c is used to coordinate the creation of the consistent checkpoints. We assume that P_c communicates with each ordinary process through reliable FIFO channel. The messages transmitted between ordinary processes are referred to as computation messages, and the messages transmitted between coordinating process and ordinary process are referred to as system messages. In order to ensure correct computation, if P_i sends computation messages to P_j, P_j must compute the computation messages from P_i according to the sending order.

Each checkpoint taken by a process is assigned a unique sequence number. The checkpoint sequence number of the process P_i is denoted by csn_i. The j-th (j>0) checkpoint of process P_i is assigned a sequence number j and csn_j is set to j. The j-th checkpoint interval[6] of process P_i denotes all the computation performed between its j-th and (j+1)-th checkpoint, including the j-th checkpoint but not the (j+1)-th checkpoint.

Each computation message sent by P_j is assigned a sequence number. The sequence number of each computation message is denoted by mid. In i-th (i>=0)checkpoint interval of P_j, the mid of first computation message sent to P_k (k≠j) is set to 1, and the mid of subsequent computation message sent to P_i increases monotonically. mid of a computation message m is denoted by m.mid.

2.2. Checkpoints Creation

Chandy and Lamport [7] formally defined the concept of a consistent distributed system state. Briefly, a consistent distributed system state is formed by a set of process states. A checkpoint is a saved state of a process. A set of checkpoints, one per process in the system, is consistent if the saved states form a consistent distributed system state.
Our algorithm saves two types of checkpoints on stable storage: tentative and permanent. A permanent checkpoint can’t be undone, and a tentative checkpoint can be undone or changed to a permanent checkpoint.

Each ordinary process $P_i$ only computes the effective computation messages in the received messages. A computation message $m_i$ is a effective computation message if and only if $m_i$ is first received by $P_i$.

Definition 1. Suppose $P_1$, $P_2$, ..., $P_n$ denote all ordinary processes in a distributed system; message sent set of $P_i$ in $h_{th}$ checkpoint interval is defined as:

$$MS_i = \{ MS_{i1}, MS_{i2}, ..., MS_{in}\}, i=1, 2, ..., n$$

Where, $MS_{ij}$ ($i \neq j$)denotes the set of the messages that $P_i$ sends to $P_j$ in $h_{th}$ checkpoint interval.

Definition 2. Suppose $P_1$, $P_2$, ..., $P_n$ denote all ordinary processes in a distributed system; message received set of $P_i$ in $h_{th}$ checkpoint interval is defined as:

$$MR_i = \{ MR_{i1}, MR_{i2}, ..., MR_{in}\}, i=1, 2, ..., n$$

Where, $MR_{ij}$ ($i \neq j$)denotes the set of the messages that $P_i$ receives from $P_j$ in $h_{th}$ checkpoint interval.

We assume that $MR_{ij}.Mid$ denotes the maximum mid of the messages in $MR_{ij}$.

Definition 3. Suppose $P_1$, $P_2$, ..., $P_n$ denote all ordinary processes in a distributed system; message computed set of $P_i$ in $h_{th}$ checkpoint interval is defined as:

$$MC_i = \{ MC_{i1}, MC_{i2}, ..., MC_{in}\}, i=1, 2, ..., n$$

Where, $MC_{ij}$ ($i \neq j$)denotes the set of the messages that $P_i$ computes from $P_j$ in $h_{th}$ checkpoint interval.

Theorem 1. If $\forall m_k, m_k \in MS_{ij} \land m_k \in MC_{ji} (i=1, 2, ..., n; j=1, 2, ..., n; i \neq j)$, then the system has a consistent distributed system state.

Proof. Since $MS_{ij}$ ($i \neq j$)denotes the set of the messages that $P_i$ sends to $P_j$ in $h_{th}$ checkpoint interval, $MC_{ji}$ ($i \neq j$)denotes the set of the messages that $P_j$ computes from $P_i$ in $h_{th}$ checkpoint interval; If $\forall m_k, m_k \in MS_{ij} \land m_k \in MC_{ji} (i \neq j)$, which denotes that $P_j$ has computed all the messages that $P_i$ has sent. If $\forall m_k, m_k \in MS_{ij} \land m_k \in MC_{ji} (i=1, 2, ..., n; i \neq j)$, which denotes that $P_j$ has computed all the messages from other processes. If $\forall m_k, m_k \in MS_{ij} \land m_k \in MC_{ji} (i=1, 2, ..., n; j=1, 2, ..., n; i \neq j)$, which denotes that all the processes has computed all the messages from other processes. In conclusion, the system has a consistent distributed system state. So the conclusion is true.

From the meaning of computer clock, the interprocess communication is not synchronous because different computer clock is difficult to achieve synchronization. The improved vector logical clock[8,9,10] is proposed in this paper in order to better describe communication of inter-process.
Definition 4. Suppose $P_1, P_2, \ldots, P_n$ denote all ordinary processes in a distributed system; the improved vector logical clock of $P_i$ is defined as:

$$R_i = (R_{i1}, R_{i2}, \ldots, R_{in}), \ i=1, 2, \ldots, n$$

Where, $R_{ij}(i \neq j)$ is a nonnegative integer variable maintained by $P_i$. Its value is one larger than maximum mid of messages in $MC_{ij}$.

Definition 5. Suppose $P_1, P_2, \ldots, P_n$ denote all ordinary processes in a distributed system; the sending vector of $P_i$ is defined as:

$$S_i = (S_{i1}, S_{i2}, \ldots, S_{in}), \ i=1, 2, \ldots, n$$

Where, $S_{ij}(i \neq j)$ is a nonnegative integer variable maintained by $P_i$. Its value is equal to maximum mid of messages in $MS_{ij}$.

Theorem 2. If $R_{ij} = S_{ji} + 1(i \neq j)$, then the messages that are sent to $P_i$ by $P_j$ are computed by $P_i$.

Proof. Since the value of $R_{ij}$ is one larger than maximum mid of messages in $MC_{ij}$, the value of $S_{ji}$ is equal to maximum mid of messages in $MS_{ji}$, so the conclusion is true.

Theorem 3. If $R_{ij} = S_{ji} + 1(i=1,2,\ldots,n; j=1,2,\ldots, n; i \neq j)$, then the system has a consistent distributed system state.

Proof. Since the value of $R_{ij}$ is one larger than maximum mid of messages in $MC_{ij}$, the value of $S_{ji}$ is equal to maximum mid of messages in $MS_{ji}$. According to theorem 1 and theorem 2, so the conclusion is true.

The process of checkpointing is as follows: When $P_c$ initiates a checkpointing process, it propagates checkpointing request to the ordinary processes. When $P_i$ receives a checkpointing request, $P_i$ will take a tentative checkpoint if $R_{ij} = MR_{ij}.mid$ $(j=1,2,\ldots, n; i \neq j)$. If $R_{ij} = S_{ji} + 1(i=1,2,\ldots,n; j=1,2,\ldots, n; i \neq j)$, we know that the tentative checkpoints are consistent according to theorem 3; so $P_c$ informs the ordinary processes to make the tentative checkpoints permanent.

2.3. Identification of problems

In unreliable non-FIFO channel, the system can lose, duplicate, or reorder messages [5].

The relation of between a computation message $m_k$ and $MC_{ij}$ is as follows: When $P_i$ receives a computation message $m_k$ from $P_j$, if $m_k \in MS_{ji} \land m_k.mid = R_{ij}$, $P_i$ will computes $m_k$ and $m_k$ will be put into $MC_{ij}$. $R_{ij}$ adds 1 automatically. If only $\exists m_j \in MR_{ij} \land m_j \in MS_{ji} \land m_j.mid = R_{ij}$, $P_i$ will computes $m_i$ and $m_i$ will be put into $MC_{ij}$. 
We assume that cp-state \( i \) is a Boolean which is set to 1 if \( P_i \) is in the checkpointing process.

Definition 6. Suppose \( P_1, P_2, \ldots, P_n \) denote all ordinary processes in a distributed system; If
\[ \exists m_k \in (m_k \in MS_{ij} \land m_k . mid < MR_{ji} . mid \land m_k \notin MR_{ji}) \lor (m_k \in MS_{ij} \land cp-state_j = 1 \land m_k . mid \geq R_{ji}) \],
which denotes message \( m_k \) is lost.

Definition 7. Suppose \( P_1, P_2, \ldots, P_n \) denote all ordinary processes in a distributed system; \( P_j \) receives a
computation message \( m_k \) from \( P_i \). If \( m_k \in MS_{ij} \land m_k . mid > R_{ji} \land m \notin MR_{ji} \), which denotes message \( m_k \) is reordered.

Definition 8. Suppose \( P_1, P_2, \ldots, P_n \) denote all ordinary processes in a distributed system; \( P_j \) receives a
computation message \( m_k \) from \( P_i \). If \((m_k \in MS_{ij} \land \exists m_l \in MR_{ij} \land m_k . mid = m_l . mid) \lor (m_k \notin MS_{ij})\),
which denotes message \( m_k \) is a duplicate.

In Fig.1, the system has taken \((i-1)_th\) (i>=1) consistent checkpoint. \( P_1 \) sends the computation message \( m_1, m_2, m_3 \) and \( m_4 \) to \( P_2 \) in \((i-1)_th\) checkpoint interval. \( m_1 . mid, m_2 . mid, m_3 . mid \) and \( m_4 . mid \) are assigned 1, 2, 3 and 4 respectively according to the sending order of messages. \( MS_{12} = \{ m_1, m_2, m_3, m_4 \} \). Message
\( m_1 \) first resent by \( P_1 \) is denoted by \( m_1^1 \), and message \( m_1 \) resent a second time by \( P_1 \) is denoted by \( m_1^2 \).

After \( P_2 \) receives message \( m_3 \) and \( m_4 \), \( MR_{21} \) is equal to \( \{ m_3, m_4 \} \) and \( MC_{21} \) is NULL. \( MR_{21} . mid \) is
equal to 4. Now, \( m_1 \) and \( m_2 \) are in \( MS_{12} \), but \( m_1 . mid \) and \( m_2 . mid \) are less than \( MR_{21} . mid \) and \( m_1 \) and \( m_2 \) are not in \( MR_{21} \); so \( m_1 \) and \( m_2 \) are lost during delivery and \( P_2 \) will never obtain the messages.

When \( P_2 \) receives message \( m_2 \), \( MR_{21} \) is NULL and \( MC_{21} \) is NULL. \( R_{21} \) is equal to 1. Now, \( m_3 \) is in \( MS_{12} \),
but \( m_3 . mid \) is larger than \( R_{21} \) and \( m_3 \) is not in \( MR_{21} \); so \( m_3 \) is reordered. When \( P_2 \) receives message \( m_4 \),
\( MR_{21} \) is equal to \( \{ m_3 \} \) and \( MC_{21} \) is NULL. \( MR_{21} . mid \) is equal to 3 and \( R_{21} \) is equal to 3. Now, \( m_4 \) is in
\( MS_{12} \), but \( m_4 . mid \) is larger than \( R_{21} \) and \( m_4 \) is not in \( MR_{21} \); so \( m_4 \) is reordered.

After \( P_2 \) receives message \( m_2^2 \), \( MR_{21} \) is equal to \( \{ m_2^2, m_3, m_4 \} \). Because \( m_2^2 . mid \) is equal to \( R_{21} \), so
\( P_2 \) computes message \( m_1^2 \) and \( m_2^2 \) is put into \( MC_{21} \). \( R_{21} \) adds 1 automatically. After \( P_2 \) receives message \( m_2^2 \),
\( MR_{21} \) is equal to \( \{ m_2^2, m_2, m_3, m_4 \} \). Because \( m_2^2 . mid \) is equal to \( R_{21} \), so \( P_2 \) computes message \( m_2^2 \) and
\( m_2^2 \) is put into \( MC_{21} \). \( R_{21} \) adds 1 automatically. \( P_2 \) computes message \( m_3 \) and \( m_3 \) is put into \( MC_{21} \) because
\( m_3 . mid \) is equal to \( R_{21} \) and \( m_3 \) is in \( MR_{21} \). It is the same with message \( m_4 \). When \( P_2 \) receives message \( m_2^1 \),
\( MR_{21} \) is equal to \( \{ m_1^1, m_2^1, m_3, m_4 \} \). Because \( m_2^1 . mid \) is equal to \( m_2^2 . mid \), so \( m_2^1 \) is a duplicate. When
\( P_2 \) receives message \( m_1^1 \) in \( i_{th} \) checkpoint interval, because \( m_1^1 \) is sent by \( P_1 \) in \((i-1)_{th}\) checkpoint interval, so
\( m_1^1 \) is a duplicate.

In order to take a consistent set of checkpoints, our coordinated checkpointing algorithm must resolve message losses, message reordering and messages duplicate. The reason of livelocks [3] is that a process
receives the same computation message twice when the process rollback recovery. We can resolve the livelocks by using the measure of resolving the messages duplicate.

### 2.4. Handling the problems

In order to get correct computation and guarantee a correct recovery following a failure, we must take a consistent set of checkpoints. So we should ensure that the above-mentioned problems are resolved correctly.

#### 1. Message Losses

Message losses is defined that some messages are lost during delivery. Message losses can lead to the incorrect computation result and inconsistency. We let ordinary processes resend the lost computation messages to resolve the message losses. So we must save the determinants of each computation message on the stable storage of the sender process.

Definition 9. Suppose \( P_1, P_2, \ldots, P_n \) denote all ordinary processes in a distributed system; the set of sending lists is defined as:

\[
SQ_i = \{SQ_{i1}, SQ_{i2}, \ldots, SQ_{in} \}, i=1, 2, \ldots, n
\]

Where, \( SQ_{ij} \) is a list of records maintained by each process \( P_i \) for sending the computation message to \( P_j \) in \( k_{th} (k\geq 0) \) checkpoint interval. Each record has the following fields: Mid and Contents. Mid is the mid of the sent message. Contents is the contents of the sent message. \( SQ_{ij}[k] \) is the \( k_{th} \) record of \( P_i 's \) \( SQ_{ij} \) list;

The process \( P_i \) will save the determinants of message \( m_k \) to \( SQ_{ij}[k] \) on the stable storage after process \( P_i \) sends a computation message \( m_k \) to \( P_j \) in \( b_{th} (b\geq 0) \) checkpoint interval. \( SQ_{ij}[k] \).Mid and \( SQ_{ij}[k] \).Contents are \( k \) and \( m_k \) respectively. \( P_j \) will send resending message request when \( P_j \) checks that message \( m_l \) from \( P_i \) is lost. \( P_i \) receives the resending message request and resend the message \( m_l \) saved in \( SQ_{ij} \) to \( P_j \).

In order to make more efficient use of stable storage, each process \( P_i \) will empty \( SQ_{ij} \) (\( j=1,2,\ldots,n \)) if \( (b+1)_{th} \) consistent checkpoint is taken.

#### 2. Message Reordering

Message reordering is defined that some messages are reordered. If we compute the messages according the receiving order, the system may lead to the incorrect result. In order to resolve the message reordering, we must let each process computes the messages from the same process according to sending order.

Definition 10. Suppose \( P_1, P_2, \ldots, P_n \) denote all ordinary processes in a distributed system; the set of receiving lists is defined as:

\[
RQ_i = \{RQ_{i1}, RQ_{i2}, \ldots, RQ_{in} \}, i=1, 2, \ldots, n
\]

Where, \( RQ_{ij} \) is a list of records maintained by each process \( P_i \) for saving the reordered messages from \( P_j \) in \( k_{th} (k\geq 0) \) checkpoint interval, where each record has the following fields: Mid and Contents. Mid is the mid of
the received message. Contents is the contents of the received message. \( RQ_{ij}[k] \) is the \( k^{th} \) record of \( P_i \)'s \( RQ_{ij} \) list;

The process \( P_i \) receives a computation message \( m_k \) \((k>1)\) from \( P_j \). If message \( m_{k-1} \) has not been computed, message \( m_k \) is reordered. So process \( P_i \) will save the determinants of message \( m_k \) to \( RQ_{ij}[n](n>=1) \) on the stable storage. If message \( m_{k-1} \) has been computed, \( P_i \) will compute message \( m_k \). If only message \( m_{k+1} \) is saved in \( RQ_{ij}[m](m>=1) \), \( P_i \) computes the message \( m_{k+1} \) got from \( RQ_{ij}[m] \) and remove the \( RQ_{ij}[m] \) from \( RQ_{ij} \).

3. Message Duplicate

When a process \( P_i \) receives a computation message \( m_k \) from \( P_j \), \( P_i \) will detect whether the message \( m \) is a duplicate. In our algorithm, when \( m \) is a duplicate message, we will drop the message.

3. A Checkpointing Algorithm Based Unreliable Non-fifo Channels

We suppose that the coordinate process \( Pc \) initiates the checkpointing process every a fixed time; and suppose that the checkpointing process must be finished in a fixed time. If the checkpointing process is not finished in the fixed time, the checkpoints can not be taken and the algorithm exits because of timeout.

3.1. The Notations and The Data Structures

The following notations and data structures are used in our algorithm:

- \( cp-state_i \): A Boolean which is set to 1 if \( P_i \) is in the checkpointing process.
- \( csn_i \): checkpoint sequence numbers (csn) at each process \( P_i \).
- \( minMid_i[j] \): A nonnegative integer variable maintained by \( P_i \). Its value is equal to minimum mid of messages from \( P_j \) that were saved in \( RQ_{ij} \) by \( P_i \).
- \( scount_i[j] \): A nonnegative integer variable maintained by \( P_j \). Its value is equal to the number of records in \( SQ_{ij} \).
- \( rcount_i[j] \): A nonnegative integer variable maintained by \( P_i \). Its value is equal to the number of records in \( RQ_{ij} \).
- \( request \): It has three fields:
  - \( P_d \): the identification of a process;
    - Min: Its value is equal to the minimum mid of messages that should be resent;
    - Max: Its value is one larger than the maximum mid of messages that should be resent;
  - If \( P_d \) = NULL, the request denotes checkpointing request, otherwise the request denotes resending message request.
  - reply: It is set to 1 if ordinary processes can make the tentative checkpoints permanent; otherwise it is set to 0 if all ordinary processes should undo the tentative checkpoints and the algorithm exits because of timeout.
A Checkpointing Algorithm Based Unreliable Non-FIFO Channels

cp-state\_i, csn\_i, S\_ij, minMid\_i[j], scount\_i[j] and rcount\_i[j] of P\_i are initialized to 0. SQ\_ij and RQ\_ij of P\_i are initialized to NULL. R\_ij of P\_i is initialized to 1.

3.2. Checkpointing Algorithm

In this section, we present our blocking checkpointing algorithm.

1. Checkpointing Initiation

The coordinator P\_c can initiate a checkpointing process. When P\_c initiates a checkpointing process, it propagates checkpointing request to the ordinary processes.

2. Reception of a request message

A process P\_i receives a request from the coordinator P\_c. If cp-state\_i=0 \land request\_P\_d=NULL, the request is a checkpointing request; otherwise the request is a resending message request.

When the request is a checkpointing request, cp-state\_i will be set to 1 and P\_i sends S\_i and R\_i to coordinator P\_c. If RQ\_i=NULL, which denotes that all the computation messages received by P\_i has been computed, P\_i will take a tentative checkpoint.

When the request is a resending message request, P\_i will resend the messages whose mid is equal to or larger than request.min and less than request.max to the process request\_P\_d.

3. Sending a Computation Message

When process P\_i sends a computation message to process P\_j, it will attach the following information: mid and csn\_j.

4. Receiving a Computation Message

When process P\_i receives a computation message from process P\_j, it will first check if rec-mid=R\_ij \land rec-csn\_j=csn\_j. If so, P\_i will compute the message and increase R\_ij. And then it check if the message whose mid is equal to R\_ij is saved in the RQ\_ij until RQ\_ij is NULL or the message whose mid is equal to R\_ij is not saved in the RQ\_ij. If so, P\_i gets the message from RQ\_ij, then P\_i computes and removes the message from RQ\_ij. P\_i increases R\_ij and minMid\_i[j] is set to the minimal mid of the messages in RQ\_ij. If RQ\_ij=NULL, minMid\_i[j] is set to 0; otherwise minMid\_i[j] is set to the minimal mid of the messages in RQ\_ij.

P\_i will drop the message if the message whose mid is rec-mid has been saved in the RQ\_ij. If rec-mid<minMid\_i[j] \land rec-mid>R\_ij \land rec-csn\_j=csn\_j, P\_i saves the message in the RQ\_ij and minMid\_i[j] is set to rec-mid. P\_i sends a resending request message to P\_c in order to inform P\_j to resend the messages whose mid...
is equal to or larger than $R_{ij}$ and less than $\text{minMid}_i[j]$. If $\text{rec-mid} > \text{minMid}_i[j] \wedge \text{rec-csn}_j = \text{csn}_i$, $P_i$ saves the message in the $RQ_{ij}$.

After process $P_i$ finishes the above actions, it will check if $\text{cp-state}_i$ is equal to 1. If $\text{cp-state}_i$ is equal to 1, $P_i$ will take the tentative checkpoint if $RQ_{ij}$ is NULL.

5. Actions in the second phase for the coordinator $P_c$

$P_c$ receives $R_i$ and $S_i$ of each process $P_i$. If $R_{ij} \neq S_{ji} + 1$, $P_c$ will inform $P_i$ to resend the messages whose mid is equal to or larger than $R_{ij}$ and less than $S_{ji} + 1$. For each process $P_i$, $P_c$ will inform $P_i$ to make its tentative checkpoint permanent if $R_{ij} = S_{ji} + 1$. When time is timeout, $P_c$ will inform each process $P_i$ to cancel its tentative checkpoint.

3.3. Algorithm Description

Actions taken when $P_i$ sends a computation message to $P_j$:

if $\text{cp-state}_i = 0$ then

send($P_i, P_j, \text{message}, \text{mid}, \text{csn}_i$);

$S_{ij} := \text{mid}; \text{scount}_i[j] := \text{scount}_i[j] + 1$;

$SQ_{ij}[\text{scount}_j].\text{mid} := \text{mid}$;

$SQ_{ij}[\text{scount}_j].\text{contents} := \text{message}$;

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

receive($P_j, P_i, \text{message}, \text{rec-mid}, \text{rec-csn}_j$);

if $\text{rec-mid} = R_{ij} \wedge \text{rec-csn}_j = \text{csn}_i$ then

compute the message;

$R_{ij} := R_{ij} + 1$;

while $R_{ij} = \text{minMid}_i[j]$ do

$\text{temp} := 1$;

while $\text{temp} \leq \text{rcount}_i[j] \wedge RQ_{ij}[\text{temp}].\text{mid} \neq R_{ij}$ then

$\text{temp} := \text{temp} + 1$;

Get the message from $RQ_{ij}[\text{temp}]$.

Compute the message;

Remove $RQ_{ij}[\text{temp}]$ from $RQ_{ij}$;

$\text{rcount}_i[j] := \text{rcount}_i[j] - 1$; $R_{ij} := R_{ij} + 1$;

if $RQ_{ij}$ = NULL then
\[
\text{minMid}_i[j] = 0;
\]
else
\[
\text{temp} := 1; \quad \text{minMid}_i[j] := \text{RQ}_{ij}[	ext{temp}].\text{mid};
\]
while \(\text{temp} \leq \text{rcount}_i[j]\) then
\[
\text{If } \text{RQ}_{ij}[	ext{temp}].\text{mid} < \text{minMid}_i[j] \text{ then }
\]
\[
\text{minMid}_i[j] := \text{RQ}_{ij}[	ext{temp}].\text{mid};
\]
\[
\text{Temp} := \text{temp} + 1;
\]
else
\[
\text{if rec-mid} < \text{minMid}_i[j] \land \text{rec-mid} > R_{ij} \land \text{rec-csn}_j = \text{csn}_i \text{ then }
\]
\[
\text{minMid}_i[j] := \text{rec-mid}; \quad \text{rcount}_i[j] := \text{rcount}_i[j] + 1;
\]
\[
\text{RQ}_{ij}\left[\text{rcount}_i[j]\right].\text{mid} = \text{rec-mid};
\]
\[
\text{RQ}_{ij}\left[\text{rcount}_i[j]\right].\text{contents} = \text{message};
\]
Send\(P_i, P_j, R_{ij}, \text{minMid}_i[j]\);
else
\[
\text{if rec-mid} > \text{minMid}_i[j] \land \text{rec-csn}_j = \text{csn}_i \text{ then }
\]
\[
\text{if } \text{RQ}_{ij} \neq \text{NULL} \text{ then }
\]
\[
\text{temp} := 1;
\]
\[
\text{While } \text{temp} \leq \text{rcount}_i[j] \land \text{RQ}_{ij}[	ext{temp}].\text{mid} \neq \text{rec-mid} \text{ then }
\]
\[
\text{temp} := \text{temp} + 1;
\]
\[
\text{If } \text{temp} > \text{rcount}_i[j] \text{ then }
\]
\[
\text{rcount}_i[j] := \text{rcount}_i[j] + 1;
\]
\[
\text{RQ}_{ij}\left[\text{rcount}_i[j]\right].\text{mid} = \text{rec-mid};
\]
\[
\text{RQ}_{ij}\left[\text{rcount}_i[j]\right].\text{contents} = \text{message};
\]
else
\[
\text{Drop the message;}
\]
else
\[
\text{drop the message;}
\]
if cp-state\(_i = 1) \text{ then }
\[
\text{if } \text{RQ}_i = \text{NULL} \text{ then }
\]
\[
\text{if tckp}_i = 1 \text{ then }
\]
\[
\text{undo the tentative checkpoint;}
\]
\[
\text{tckp}_i := 0;
\]
\[
\text{send(U}_i, \text{T}_i, \text{mark}_i);
\]
\[
\text{take tentative checkpoint;}
\]
\[
\text{tckp}_i := 1;
\]
Actions at process $P_c$, on receiving a resending message request from $P_i$:

receive($P_i$, $P_j$, $R_{ij}$, $\text{minMid}_{ij}[j]$);
request. $P_d := P_j$; request.min := $R_{ij}$;
request.max := $\text{minMid}_{ij}[j]$;
Send($P_i$, request);

Actions in the first phase for the coordinate process $P_c$:

request. $P_d := \text{NULL}$;
for $i := 1$ to $N$ do
  send($P_i$, request);

Actions at process $P_i$, on receiving a request from $P_c$:

receive($P_i$, rec-request);
if $\text{cp-state}_i = 0 \land \text{rec-request}.P_d = \text{NULL}$ then
  $\text{cp-state}_i := 1$;
  send($S_i, R_i$);
if $RQ_i = \text{NULL}$ then
  take tentative checkpoint;
else
  $k := \text{rec-request.min}$;
  while $k < \text{rec-request.max}$ do
    temp:= 1;
    While temp <= $\text{scount}_i[\text{rec-request}.P_d] \land \text{SQ}[\text{rec-request}.P_d][\text{temp}] \neq k$ do
      Temp:=temp+1;
      Get the message from $\text{SQ}[\text{rec-request}.P_d][\text{temp}]$;
    send($P_i$, rec-request,$P_d$, message, $k$, $\text{csn}_i$);
    $k := \text{k+1}$;

Actions in the second phase for the coordinate process $P_c$:

receive($S_i, R_i$);
num:=num+1; ack:=0;
if num=N then
  for $i := 1$ to $N$ do
    tag:=0;
  for $j := 1$ to $N$ do
    if $R_{ij} \neq S_{ji} + 1$ then
      request. $P_d := P_j$;
      request.min := $R_{ij}$;
      request.max := $S_{ji} + 1$;
send(P_i, request);
tag:=1
if tag=0 then
  ack:=ack+1;
while ack<N do
  while not timeout do
    receive(S_n, R_n, rec-mark_n);
    if mark_c=rec-mark_n then
      Tag:=0;
      for j:=1 to N do
        if R_nj \neq S_jn+1 then
          request.Pd:= P_j; request.min:= R_nj;
          request.max:= S_jn+1; send(P_i, request);
tag:=1
          if tag=0 then
            ack:=ack+1;
          if timeout then
            reply:=0; mark_c:= mark_c+1;
            for i:=1 to N do
              send(P_i, reply);
            exit the algorithm;
          reply:=1; mark_c:=0;
          for i:=1 to N do
            send(P_i, reply);
        Actions at other process P_i on receiving a reply message:
        receive(P_i, reply);
        if reply=0 then
          cp-state_i:=0; mark_i:=mark_i+1;
        else
          make the tentative checkpoint permanent;
          mark_i:=0; csn_i:=csn_i+1; cp-state_i:=0; tckp_i:=0;
          scount_i:=0; rcount_i:=0; S_i:=0; R_i:=0; SQ_i:=NULL;

4. Algorithm Analysis

Theorem 4. The algorithm can create consistent global checkpoints.

Proof. When P_c initiates a checkpointing process, it propagates checkpointing request to the ordinary
processes. Each process P_i will send S_i and R_i to P_c, and then P_i takes a tentative checkpoint if RQ_i is NULL.
If \( R_{ij} = S_{ji} + 1 \) (\( i = 1, 2, \ldots, n; j = 1, 2, \ldots, n; i \neq j \)), which denotes the computation messages sent by all the sender process have been computed by their own receiver process. \( P_c \) informs each process \( P_i \) to make its tentative checkpoint permanent. Now, these checkpoints are consistent. If \( R_{ij} \neq S_{ji} + 1 \) (\( i = 1, 2, \ldots, n; j = 1, 2, \ldots, n; i \neq j \)), \( P_c \) will inform \( P_j \) to resend the lost messages to \( P_i \) until \( R_{ij} \) is equal to \( S_{ji} + 1 \). The algorithm will exit and undo the tentative checkpoints if time is timeout. In conclusion, the checkpoints created by our algorithm are consistent global checkpoints.

Theorem 5. Each process can compute the messages correctly.

Proof. Our algorithm ensures that each process computes the messages from the same process according to their sending order. When some messages are lost, the algorithm will let the sender process resend the lost messages in order that all the messages can be computed. When a process receives a duplicate message, the process don’t computes the message in order that each process computes the message only once. In conclusion, each process can compute the messages correctly.

We assume that \( n \) is the number of processes; \( m \) is the number of lost messages before checkpointing phase; \( h \) is the number of lost messages and \( t \) is the number of processes that lost messages in checkpointing phase. Before checkpointing phase, process \( P_i \) checks that a computation message is lost and it inform \( P_c \). \( P_c \) inform the sender process to resend the lost message. In checkpointing phase, \( P_c \) sends checkpointing request to each process and each process sends a system message to \( P_c \). Eventually, \( P_c \) needs to send a reply to each process. \( P_c \) will inform the sender processes to resend the lost messages if \( h \) is not equal to 0; and then the receiver processes need to send a system message to \( P_c \). So the number of system messages is \( O(3n + 2m) \) if \( h \) is equal to 0. The number of system messages is \( O(2n + 2m + h) \) if \( h \) is not equal to 0.

5. Conclusion

In this paper, we propose a coordinated checkpointing algorithm based unreliable non-FIFO channel. In unreliable non-FIFO channel, the system can lose, duplicate, or reorder messages. The processes may not compute some messages because of message losses; the processes may compute some messages twice or more because of message duplicate; the processes may not compute messages according to their sending order because of message reordering. The above-mentioned problems make processes produce incorrect computation result, consequently, prevent processes from taking consistent global checkpoints. Our algorithm assigns each message a sequence number in order to resolve above-mentioned problems. During the establishing of the checkpoint, the consistency of checkpoint can be determined by the sequence number of sending and receiving messages. We can identify the lost messages, reordering messages and duplicate messages by checking the sequence number of sending and receiving messages. We resolve above-mentioned problems by resending the lost messages, buffering the reordering messages and dropping the duplicate messages. Our algorithm makes processes take consistent global checkpoints.

References


