Checkpoint and Rollback in Asynchronous Distributed Systems

Hiroaki Higaki  Kenji Shima  Takayuki Tachikawa  Makoto Takizawa

Department of Computers and Systems Engineering
Tokyo Denki University
{hig,sima,tachi,taki}@takilab.k.dendai.ac.jp

Abstract

This paper proposes a novel algorithm for taking checkpoints and rolling back the processes for recovery in asynchronous distributed systems. The algorithm has the following properties: (1) Multiple processes can simultaneously initiate the checkpointing. (2) No additional message is transmitted for taking checkpoints. (3) A set of local checkpoints taken by multiple processes denotes a consistent global state. (4) Multiple processes can initiate simultaneously the rollback recovery. (5) The minimum number of processes are rolled back. (6) Each process is rolled back asynchronously. The number of messages for rolling back the processes is $O(l)$ where $l$ is the number of channels. Therefore, the system is kept highly available by the algorithm presented in this paper.

1 Introduction

Information systems become distributed and are getting larger by including various kinds of component systems and interconnecting with various systems, e.g., by the Internet, in the world. The distributed systems are designed and developed by using widely available products including free softwares rather than specially designed hardwares and softwares. Distributed applications are realized by cooperation of multiple processes executed in multiple computers. These components are not always guaranteed to support enough reliability and availability for the applications. It is critical to discuss how to make and keep the systems so reliable and available that even fault-tolerant applications could be computed in the systems.

Checkpointing and rollback recovery are well-known time-redundant techniques to allow processes to make progress even if some processes fail. The processes take checkpoints by saving their state information in the local logs while executing the applications. If the processes fail in the system, the processes are rolled back to the checkpoints by restoring the saved state information and then the applications are restarted to be executed from the checkpoints. In this paper, we assume that every failure is transient, e.g., hardware errors, process crashes, transaction aborts, and communication deadlocks. The failures are unlikely to recur after the processes are restarted.

We have to consider how to keep the system consistent when taking checkpoints and rolling back the processes. The consistency of the global state is formalized by Chandy and Lamport [3]. Many papers [2, 3, 5, 6, 7, 9, 10, 11, 12, 13, 14] have discussed so far how to take the consistent checkpoints among multiple processes. In addition, we have to discuss how to roll back the processes for recovery if some process fails. If each process is rolled back independently of the other processes, the system state may be inconsistent. One idea [7] is to synchronize all the processes to be rolled back by using the protocols similar to the two-phase commitment protocol [1]. However, it takes time to exchange messages among the processes. In this paper, we would like to discuss a new method where the processes are allowed to be asynchronously rolled back and restarted.

In section 2, the conventional checkpointing and rollback recovery methods are reviewed. In section 3, we show a basic algorithm for taking checkpoints and rolling back processes. In section 4, we make clear how livelocks occur in the rollback recovery. A livelock-free algorithm is proposed in section 5. The evaluation of the algorithm is presented in section 6.

2 Checkpoint and Rollback

2.1 Consistent state

A distributed system is composed of multiple processes interconnected by channels, i.e., $(V, L)$ where $V = \{p_1, \ldots, p_n\}$ is a set of processes and $L \subseteq V^2$ is a set of channels. $(p_i, p_j) \in L$ indicates a channel from $p_i$ to $p_j$. In the distributed system, three kinds of events occur: message-sending, message-receiving and local events. A state of the process is changed when an event occurs. An event $\epsilon$, happens before $\epsilon'$ ($\epsilon \Rightarrow \epsilon'$) iff one of the following conditions is satisfied [8]:

- $\epsilon$ occurs before $\epsilon'$ in the same process.
- $\epsilon$ is a message-sending event for a message $m$ and $\epsilon'$ is a message-receiving event for $m$. 


2.2 Checkpointing

There are two approaches to taking checkpoints among multiple processes: asynchronous and synchronous checkpointing. In the asynchronous checkpointing [2, 5, 13, 14], the processes take the checkpoints without cooperating with the other processes. This approach implies less overhead because it requires no communication among the processes. However, domino effects may occur [11]. On the other hand, in the synchronous checkpointing [3, 6, 7, 11, 12], multiple processes are coordinated to take the checkpoints by using the protocols similar to the two-phase commit protocol [1]. Here, the overhead for taking checkpoints is higher than the asynchronous checkpointing while no domino effect occurs. This paper discusses the synchronous checkpointing.

In the conventional checkpointing [3, 5, 7, 11, 12, 13, 14], if some process takes a local checkpoint, all the processes in the system are required to take local checkpoints. Moreover, in the conventional synchronous checkpointing, additional messages to take the checkpoints are transmitted and the processes are suspended during the checkpointing procedure. However, all the processes are not always needed to take local checkpoints to keep the system consistent after the rollback recovery. We would like to discuss which processes have to take the local checkpoints. We define a semi-consistent global state.

Definition (semi-consistent) For a distributed system \(\{V, L\}\), let \(P\) be a subset of \(V\). A global state \(S\) is semi-consistent for \(P\) iff there is no orphan message for every channel of each process in \(P\).

The system is kept consistent after the rollback recovery iff a global state \(S\) is semi-consistent for a subset \(P\) of processes and only and all the processes in \(P\) are rolled back. Here, suppose that a process \(p_i\) takes a checkpoint \(c^i\). If \(p_i\) sends a message \(m\) to \(p_j\) after taking \(c^i\), \(m\) is referred to as a checkpoint message of \(c^i\) to \(p_j\). In our checkpointing algorithm, \(m\) contains the information on whether \(p_i\) takes \(c^i\) before sending \(m\) or not. In order that a global checkpoint denotes a semi-consistent global state, \(p_i\) takes a checkpoint \(c^i\) if the following condition is satisfied:

**Checkpoint** If a message-receiving event \(e\) for a checkpoint message \(m\) occurs in a process \(p_j\) and \(p_j\) does not take a local checkpoint \(c^j\), \(p_j\) takes \(c^j\) just before \(e\). □

This means that \(p_j\) can newly take a local checkpoint after \(p_j\) is rolled back but cannot if \(p_j\) had taken a local checkpoint. By using this checkpointing algorithm, the minimum number of processes take checkpoints and no additional message is transmitted to take the checkpoints.

2.3 Rollback

In the conventional rollback algorithms [3, 5, 7, 11, 12, 13, 14], the processes have to be synchronized to be restarted by the following procedure:
1) Request messages are transmitted from the coordinator process to all the other processes called cohort processes.

2) Each cohort is rolled back to the checkpoint and a reply message is transmitted from each cohort to the coordinator.

3) The coordinator transmits restart messages to all the cohorts. Each cohort is restarted from the checkpoint.

One of the disadvantages of the algorithms is that all the processes are suspended and additional messages are transmitted to synchronize all the processes. The larger the system becomes, the longer the processes are suspended. Thus, the system becomes less available. In order to keep the system highly available with the rollback recovery, we would like to discuss a method where the processes are asynchronously restarted from the checkpoints.

3 Basic Algorithm

In this section, we would like to show a basic algorithm for taking checkpoints and rolling back processes by using an example shown in Figure 2. The system consists of four processes \{p_1, p_2, p_3, p_4\} fully connected by bidirectional channels. Each process \(p_i\) takes a checkpoint \(c_i\). If some process fails, \(p_i\) is rolled back to \(c_i\) and restarted from \(c_i\). \(c_i\) represents the \(i\)th checkpoint taken by \(p_i\). \(r_i\) represents a possible rollback event when \(p_i\) is rolled back to \(c_i\). \(c_i\) is active if \(p_i\) takes \(c_i\) and \(r_i\) does not occur. For example, \(c_2\) is active when \(p_1\) sends \(m_1\). After \(r_3\) occurs, \(p_1\) is restarted from \(c_2\), while \(c_2\) is not active.

If \(p_i\) fails, it is not sufficient to restart \(p_i\) from an active checkpoint \(c_i\) because there may be orphan messages. For example, if \(r_1\) occurs in \(p_1\) and \(p_1\) is restarted from \(c_1\), \(m_3\) is an orphan message. In order to obtain a semi-consistent global state after the rollback recovery, \(p_2\) has to be restarted from \(c_2\). That is, if \(p_1\) is restarted from \(c_2\), an event \(c_i\) in \(p_2\) where \(c_i \rightarrow c_2\) has to be canceled by a rollback recovery in \(p_2\).

Even if \(p_1\) and \(p_2\) are rolled back and restarted from \(c_2\) and \(c_2\), respectively, \(p_3\) and \(p_4\) are not required to be rolled back. That is, if no event \(c_i\) occurs in \(p_1\), \(p_2\) is not required to be rolled back. Thus, if \(p_1\) is rolled back, we have to identify which processes are required to be rolled back in the system.

First, we would like to define the precedence relation among the active checkpoints.

Definition (checkpoint precedence) Let \(c_i\) and \(c_j\) be active checkpoints taken by processes \(p_i\) and \(p_j\), respectively. Let \(c_i \rightarrow c_j\) be events such that \(c_i \rightarrow c_j\) and \(c_j \rightarrow c_i\) precedes \(c_i \rightarrow c_j\) if \(c_i \rightarrow c_j\). □

In Figure 2, when \(p_2\) receives \(m_3\), \(c_2 \rightarrow c_3\) and \(c_4 \rightarrow c_2\) are not included in \(D(p_2)\). □

Definition (rollback domain) A rollback domain \(D(p_i)\) of a process \(p_i\) is defined to be a following subset of processes in the system:

1) \(p_i \in D(p_i)\) if there is an active checkpoint \(c_i\) in \(p_i\). Otherwise, \(D(p_i) = \emptyset\).

2) \(p_j \in D(p_i)\) if \(c_j\) is active in \(p_j\) and \(c_j \rightarrow c_i\) or \(c_i \rightarrow c_j\) where \(c_i\) is active in \(p_i\).

3) Only the processes satisfying 1) and 2) are included in \(D(p_i)\). □

In Figure 2, \(m_3\) is transmitted from \(p_1\) to \(p_2\), \(p_2\) takes \(c_1\) before accepting \(m_3\). Here, \(D(p_1) = D(p_2) = \{p_1, p_2\}\).

For each \(p_i \in D(p_1)\), it is clear that \(p_i \in D(p_1)\) and \(D(p_1) \cap D(p_2) = \emptyset\) if \(p_i \notin D(p_1)\). A set \(C = D(p_1)\) of processes is referred to as a rollback class. According to the definition, two different rollback classes are disjoint. Each time a message-sending or message-receiving event occurs in a process \(p_i\), a rollback class \(C_i\) of \(p_i\) is changed as follows.

Change of rollback class If \(p_i\) in a rollback class \(C_i\) is changed to be in \(C_{i+1}\) on sending a message \(m\) and \(p_j\) in \(C_j\) is changed to be in \(C_{j+1}\) on receiving \(m\) from \(p_i\), then \(C_{i+1} = C_i \cup C_j\). □

In Figure 2, before \(m_3\) is transmitted, \(D(p_1) = D(p_2) = C = \{p_1, p_2\}\) and \(D(p_3) = D(p_4) = C' = \{p_3, p_4\}\). When \(m_3\) is transmitted from \(p_2\) to \(p_3\), \(C\) and \(C'\) are changed to \(C'' = C \cup C' = \{p_1, p_2, p_3, p_4\}\).

For every state \(S\) of a system \((V, E)\) and a subset \(V' \subseteq V\), let \(S_{V'}\) denote a projection of \(S\) into \(V'\), i.e., a collection of the local states of the processes in \(V\) and the states of the channels of the processes in \(V'\). Let \(A_{V'}\) be a set of the local states denoted by the active checkpoints taken by the processes in \(V\). If some process \(p_i \in V'\) has no active checkpoint, the local state of \(p_i\) in \(S\) is included in \(A_{V'}\).

Theorem 1 For every rollback class \(C\) at every system state \(S\), \(S_{V' \subseteq V'} \cap A_{V'}\) is semi-consistent for \(V'\) if 1) \(C \subseteq V'\) and 2) \(V' \cap C = \emptyset\) or \(C'\) for every rollback class \(C' \neq C\). □

That is, if a process \(p_i\) in a rollback class \(C\) fails, the system state is semi-consistent for \(C\) if all the pro-
processes in C are rolled back to the active checkpoints. This also means that C is the minimum set of processes to be rolled back for keeping the system semi-consistent after the recovery.

Suppose that $p_1$, $p_2$, $p_3$, and $p_4$ in Figure 2 are at $r_1$, $r_2$, $r_3$, and $r_4$, respectively, $p_1$ does not know that $p_3$ and $p_4$ are in $D(p_1)$ while knowing that $p_2$ is in $D(p_1)$.

Thus, $p_1$ does not have the complete information on which processes are included in C.

**Definition (rollback view)** A process $p_j$ is included in a $p_i$'s rollback view $W(p_i)$ of $D(p_i)$ if $p_i$ knows $p_j \in D(p_i)$.

Thus, $W(p_i) \subseteq D(p_i)$.

Based on the view $W(p_i)$ of $p_i$, $p_i$ can be rolled back and restarted from the active checkpoint $c_i$ by using the message diffusion protocol [4]:

1. If $p_i$ fails, $p_i$ sends a rollback request $m_r$ to all the processes in $W(p_i)$.
2. On receipt of $m_r$, $p_i$ also sends $m_r$ to all the processes in $W(p_i)$.
3. If $p_i$ receives $m_r$ from all the processes in $W(p_i)$, $p_i$ is rolled back and restarted from $c_i$.

### 4 Livelock in Rollback Recovery

Let us consider the following scenario shown in Figure 3.

1. $p_2$ takes $c_2^i$ and sends $m_1$ to $p_2$. $m_1$ is a checkpoint message. Here, $c_2^i$ is active and $D(p_2) = \{p_2, p_3\}$.
2. $p_2$ receives $m_1$ and takes $c_2^i$ where the local state of $p_2$ just before receiving $m_1$ is recorded in the log. Here, $c_2^i$ is active and $D(p_2) = D(p_3) = \{p_2, p_3\}$. A set of local checkpoints $S_1 = \{c_2^i, c_3^i\}$ is semi-consistent for $C_1 = \{p_2, p_3\}$ because there is no orphan message in the channels of $p_2$ and $p_3$.
3. $p_2$ sends $m_2$ to $p_1$. $p_1$ receives $m_2$ and takes $c_1$. Here, $c_1$ is active. $D(p_1) = D(p_3) = \{p_1, p_2, p_3\}$. $p_1$ does not know that $p_3$ is in the same rollback class, i.e., $W(p_1) = \{p_1, p_2\} \subseteq D(p_1)$.
4. $p_3$ fails and is rolled back from $r_3$ to $c_3^i$. Now, since $c_3^i$ becomes inactive, $p_3$ is not in the rollback class. Here, $D(p_1) = D(p_2) = \{p_1, p_2\}$ and $D(p_3) = \emptyset$. $S_2 = \{c_1, c_2^i\}$ is semi-consistent for $C_2 = \{p_1, p_2\}$.
5. $p_1$ sends $m_3$ to $p_3$. $p_3$ receives $m_3$ and takes a new checkpoint $c_2^i$. Here, $c_2^i$ is active. $D(p_1) = D(p_2) = D(p_3) = \{p_1, p_2, p_3\}$. $W(p_1) = D(p_1) = \{p_1, p_2, p_3\}$, $W(p_2) = \{p_1, p_2\}$, and $W(p_3) = \{p_1, p_2\}$.
6. $p_2$ is rolled back from $r_2$ to $c_2^i$ because $p_3$ is rolled back at step 4). Here, $D(p_1) = D(p_2) = \{p_1, p_3\}$ and $D(p_3) = \emptyset$. $S_3 = \{c_1, c_2^i\}$ is semi-consistent for $C_3 = \{p_1, p_3\}$.
7. $p_3$ sends $m_4$ to $p_2$, $p_2$ receives $m_4$ and takes $c_2^i$. $D(p_1) = D(p_2) = D(p_3) = \{p_1, p_2, p_3\}$, $W(p_2) = D(p_3) = \{p_1, p_2, p_3\}$, $W(p_1) = \{p_1, p_2\}$, and $W(p_2) = \{p_2, p_3\}$.
8. $p_1$ is rolled back from $r_1^i$ to $c_2^i$ because $p_3$ is rolled back at step 6). Here, $D(p_1) = \emptyset$ and $D(p_2) = D(p_3) = \{p_2, p_3\}$. $S_4 = \{c_2^i, c_3^i\}$ is semi-consistent for $C_4 = \{p_2, p_3\}$.

Thus, the rollback class $C_4$ does not become empty and the rollback recovery may be continued forever, i.e. livelock occurs.

One way to resolve the livelock is to synchronize all the processes to be restarted, e.g., by using two-phase commit protocol. However, it takes time and the control messages have to be transmitted to synchronize the processes. In this paper, we would like to discuss a new method where the processes can be restarted asynchronously. Suppose that a process $p_i$ sends a checkpoint message $m_i$ at an event $e_i$ after taking a local checkpoint $c_i$ and $p_i$ sends $m_i$ at $e_i'$ after receiving $m_i$ as shown in Figure 4. Here suppose that a rollback $r_i'$ occurs in $p_i$. Since $e_i \rightarrow e_i'$ and $c_i \rightarrow e_i \rightarrow r_i'$, $p_i$ cannot receive $m_i$ after $p_i$ is rolled back from $r_i'$, i.e., $e_i'$ is canceled. This is because $p_i$ is sure that $p_j$ would be rolled back owing to the rollback of $p_i$ and the message-receiving event for $m_i$ has to be canceled. If $p_i$ receives $m_i$, $p_i$ is required to be rolled back again due to the rollback recovery in $p_j$. Thus, the livelock may occur. Hence, in Figure 3, $p_3$ is not allowed to receive $m_3$, i.e., discards $m_3$ to realize the livelock-free rollback recovery.

In order to identify the messages to be discarded, we introduce generation concept as follows:

**Definition (generation of a process)** A generation $g(p_i)$ of a process $p_i$ is given as follows:

![Figure 4: Cause of livelock.](image)
1) \( g(p_i) = 0 \) before \( r_i \) occurs in \( p_i \).
2) \( g(p_i) = s \) between \( r_i \) and \( r_{i+1} \).

**Definition (generation of an event)** A generation \( g(e) \) of an event \( e \) in a process \( p_i \) is given when \( e \) occurs as follows:

1) \( g(e) = s \) if \( p_i \) has an active checkpoint \( c^i \).
2) \( g(e) = \perp \) (unknown) if \( p_i \) has no active checkpoint.

Each time \( p_i \) is rolled back, the generation of \( p_i \) is incremented. \( g(e) \) of an event \( e \) occurring in \( p_i \) is set to \( g(p_i) \) if the checkpoint taken most recently by \( p_i \) is active. If an event \( e' \) precedes a message-sending event of a message \( m' \) in \( p_i \), \( g(e') \) is pigged back with \( m' \). If \( p_i \) receives from \( p_j \), \( g(e') \) is pigged back with \( m' \). On receipt of \( m' \), \( p_i \) knows of \( g(e') \). If \( g(e') < g(p_i) \), \( p_i \) knows that the message-sending event of \( m' \) would be canceled by the rollback recovery in \( p_j \).

In Figure 4, \( g(p_i) = s - 1 \) when \( e' \) occurs. Thus, \( g(e') = s - 1 \) is pigged back with \( m' \) and \( m'' \). When \( r_i \) occurs in \( p_i \), \( g(p_i) \) is changed to \( s \). When \( p_i \) receives \( m_j \), \( p_i \) obtains \( g(e') = s - 1 \) pigged back with \( m_j \) while \( g(p_i) = s \). Thus, \( p_i \) discards \( m_j \) because \( p_i \) detects \( g(e') < g(p_i) \).

In the succeeding section, we would like to present the detailed algorithm using vectors of the generations for preventing the livelock.

5 Algorithms

5.1 Assumptions and definitions

A distributed system \( S = (V, L) \) consists of a finite set \( V = \{p_1, \ldots, p_n\} \) of processes and a set \( L \subseteq V^2 \) of channels. We make the following assumptions on \( S \).

A1 Every channel in \( L \) is bidirectional.
A2 Every channel in \( L \) is reliable.
A3 In each channel \( (p_i, p_j) \in L \), messages are transmitted in the first-in-first-out order from \( p_i \) to \( p_j \).
A4 \( S \) is asynchronous, i.e., a maximum message transmission delay is unknown.

The following events occur in \( p_i \):

- Message-receiving: \( p_i \) takes out a message \( m \) from a channel \((p_j, p_i)\) and accepts \( m \) from \( p_j \).
- Message-sending: \( p_i \) puts a message \( m \) to a channel \((p_i, p_j)\) to send \( m \) to \( p_j \).
- Checkpoint: \( p_i \) records the local state information in the log. The \( k \)th checkpoint taken by \( p_i \) is denoted by \( c^i_k \). Initially, every \( p_i \) takes \( c^i_0 \).
- Rollback: If \( p_i \) fails at \( r \) between \( c^i_k \) and \( c^i_{k+1} \), \( p_i \) is restarted from \( c^i_k \). The rollback to occur at \( r \) is denoted by \( r_i^k \). \( r_i^k \) exists only if \( p_i \) is rolled back to \( c^i_k \).
- A flag \( m\.flag \).
- A vector clock \( m\.clock = (m\.cl_1, \ldots, m\.cl_n) \).
- A process \( p_i \) manipulates the following variables.
  - A vector clock \( c\.cl_j \) named a checkpoint clock: Each \( c\.cl_j \) shows the generation of the active checkpoint in \( p_j \) that \( p_i \) knows. That is, when an event \( e' \) occurs in \( p_j \), \( c\.cl_j \rightarrow e' \) if \( c\.cl_i \neq \perp \). \( c\.cl_i \) is incremented by one each time \( p_i \) takes a local checkpoint. Initially, \( c\.cl_i = 0 \) and \( c\.cl_i = \perp \) for \( j \neq i \).
  - A vector clock \( r\.cl_i \) named a rollback clock: Each \( r\.cl_i \) shows the generation of the rollback most recently occurring in \( p_j \) that \( p_i \) knows. That is, on receipt of a message \( m \), \( p_i \) has no active checkpoint and \( m\.cl_j \leq r\.cl_j \). \( p_i \) detects that \( m \) is canceled by the rollback recovery, \( r\.cl_j \) is updated to \( c\.cl_i \) each time a rollback occurs in \( p_j \). Initially, \( r\.cl_i = 0 \) and \( r\.cl_i = \perp \) for \( j \neq i \).
  - A flag \( m\.flag \): If \( c^i_k \) is active, \( m\.flag = True \). Otherwise, \( m\.flag = False \).
  - A set \( W \subseteq N \) of the neighbor processes of \( p_i \) included in the rollback domain \( D(p_i) \) of, i.e., \( W(p_i) \): Initially, \( W = \emptyset \).
  - A sequence \( M \) of messages received after taking the active checkpoint in \( p_i \): Initially, \( M = \emptyset \).

For a pair of vector clocks \( v^i = (v^i_1, \ldots, v^i_n) \) and \( v^j = (v^j_1, \ldots, v^j_n) \), \( max(v^i, v^j) \) is defined to be \( \langle v_1, \ldots, v_n \rangle \) where each \( v_k = v_k^i \) if \( v_k^i \neq \perp \), \( v_k = v_k^j \) if \( v_k^j = \perp \), \( v_k = max(v_k^i, v_k^j) \) otherwise.

5.2 Checkpointing

The checkpointing algorithm has to satisfy the following requirements:

R1 A process which has an active checkpoint is included in exactly one rollback class.
R2 The global state is semi-consistent for a set of checkpoints taken by the processes in each rollback class. That is, there is no orphan message.
R3 Every process \( p_i \) has the rollback view \( W^i \) on which neighbor processes are included in the same rollback class.
R4 There is no lost message.

A process \( p_i \) takes a local checkpoint \( c^i_k \) if one of the following conditions is satisfied:

C1 If \( p_i \) decides to take a local checkpoint by such a trigger as user request or timeout, \( p_i \) takes \( c^i_k \).
C2 If a message-receiving event \( e \) occurs in \( p_i \) where \( p_i \) has no active checkpoint and \( p_i \) receives a checkpoint message \( m \) transmitted from a neighbor process \( p_j \) of \( p_i \), \( p_i \) takes \( c^i_k \) just before \( e \).

C1 means that the checkpointing can be initiated independently by multiple processes. By taking the checkpoints as presented in C2, there is no orphan message in the channel \((p_i, p_j)\). Thus, R2 is satisfied.
Suppose that $p_i$ and $p_j$ have active checkpoints $c'_i$ and $c'_j$, respectively. If $p_i$ sends a message $m$ to $p_j$, $m$ is a checkpoint message, i.e., $m.flag = True$ because $flag^j = True$. When $p_j$ receives $m$, $p_j$ does not take another checkpoint event even if $m.flag = True$. As presented in the preceding section, if $p_i$ in a rollback class $C_u$ sends a checkpoint message to $p_j$ in a rollback class $C'_v$ where $C_u \cap C'_v = \emptyset$, $C'_v$ and $C_u$ are changed to $C'_{u+1}$ and $C'_{v+1}$, respectively, where $C'_{u+1} = C'_v \cup C'_u$. Thus, R1 is satisfied.

The system has to prevent the livelock caused by the rollback recovery as discussed in the previous section. Here, we would like to present the checkpoints algorithm for the livelock-free rollback recovery. Suppose that $p_i$ and $p_j$ are in the same rollback class $C$ where $p_i$ and $p_j$ have active checkpoints $c'_i$ and $c'_j$, respectively, and $p_i$ receives a checkpoint message $m$ from $p_j$. Let $c'(m)$ denote the message-receiving event of $p_i$ and $c''(m)$ denote the message-sending event of $m$ in $p_i$. If $c''(m)$ occurs before $c'(m)$, $p_i$ discards $m$ because $p_i$ knows that $c''(m)$ is canceled eventually. In order to discard $m$, $p_i$ uses $c.CL^i$, $r.CL^i$ and $m.clock$. Each time $p_j$ sends $m$, $m.clock = (m.cl[1], \ldots, m.cl_n)$ where $m.cl_k = c.cl^j_k (k = 1, \ldots, n)$.

**Livelock-free message reception** On reception of a checkpoint message $m$ from $p_j$, $p_i$ discards $m$ if $m.cl_k \neq \bot$ and $m.cl_k \leq r.cl^j_k$ for some $k$. □

Suppose that processes $p_i$, $p_j$, and $p_k$ are in a rollback class $C$ and a checkpoint message is transmitted from $p_k$ to $p_j$. Suppose that $p_k$ is rolled back and a rollback request $m_r$ is transmitted and that $p_j$ sends a checkpoint message $m$ to $p_i$ before receiving $m_r$ and $p_i$ receives $m_r$ before $m$. Here, $r.cl^j_k$ is incremented by one if the rollback recovery occurs in $p_i$. $m.cl_k$ is the same as one given before the rollback recovery. Hence, $m.cl_k \leq r.cl^j_k$ and $m$ is discarded in $p_i$.

An identifier of a neighbor process $p_j \in N^i$ is included in the rollback view $V^j$ of $p_i$ if $p_j$ has an active checkpoint $c'_j$ and one of the following events occurs in $p_i$:

- A message-receiving event of a checkpoint message $m$ from $p_j$ and $m.cl_j \neq \bot$.
- A message-sending event of a checkpoint message $m$ to $p_j$.

Hence, $p_i$ knows that $p_i$ and $p_j$ are in the same rollback class even if no checkpoint message is transmitted between $p_i$ and $p_j$. Therefore, R3 holds.

Moreover, in order to assure that no message is lost after the rollback recovery, if $p_i$ receives a message $m$ from $p_j$ where $p_j$ has an active checkpoint $c'_j$ and $p_j$ has no active checkpoint, $p_i$ records $m$ in $M^j$. If $p_i$ is rolled back to $c'_j$ and is restarted, $p_i$ takes $m$ out of $M^j$ before receiving a message from the channels of $p_i$. By this algorithm, R4 is satisfied.

The following procedures Send($m$) and Receive($m$) is executed when a message-sending event of a message $m$ to a process $m.receiver$ and a message-receiving event of a message $m$ from a process $m.sender$ occur in $p_i$, respectively:

**Send($m$)**

$m.flag \leftarrow flag^j$;
$m.clock \leftarrow c.CL^j$;
send $m$ to $m.receiver$;

**Receive($m$)**

if $m.flag = True$
if $m.cl_j \neq \bot$
add $p_j$ to $W^i$
fi
od
$c.CL^i \leftarrow max(c.CL^i, m.clock)$;
if $flag^j = False$
$c.cl^j_k \leftarrow c.cl^j_k + 1$;
$flag^j \leftarrow True$;
checkpoint;
fi
fi
accept $m$;
fi

**5.3 Rollback recovery**

If a process $p_i$ fails, a rollback recovery procedure is initiated. The procedure is finished if the rollback class $C$ of $p_i$ becomes empty. The rollback recovery is realized by using the message diffusion protocol [4]. If $p_i$ receives the rollback request $m_r$ from $p_j$, $p_i$ sends $m_r$ to all the processes in $W^i$ except $p_j$. Then, $p_i$ restores the state information recorded at the active checkpoint $c'_j$ taken by $p_i$ and is restarted from $c'_j$. Thus, on being rolled back to the checkpoint, $p_i$ can be restarted independently of the other processes while $p_i$ has to be suspended to be synchronized with the other processes in the other algorithms [3, 7, 11, 12, 13, 14].

When $p_i$ is restarted, $r.CL^i$ is updated to be $c.CL^i$.

The following procedure Rollback() is executed when $p_i$ is recovered or $p_i$ receives a rollback request message $m_r$:

**Rollback()**

foreach $p \in W^i$ do
send a rollback request message $m_r$ to $p$;
fi
$flag^j \leftarrow False$;
$r.CL^j \leftarrow c.CL^j$;
foreach $j \neq i$ do
$c.cl^j_k \leftarrow \bot$;
fi
restart from the active checkpoint of $p_i$;
6 Evaluation

First, we would like to show the logical properties of the algorithm presented in this paper.

Theorem 2 The rollback algorithm is terminated in finite time.

Proof Suppose that a process \( p_i \) is included in a rollback class \( C \). Let \( |C| \) be the number of processes included in \( C \). Consider a case that \( p_i \) receives a checkpoint message from a process \( p_j \in C \) and then \( p_i \) is rolled back and restarted from the active checkpoint. If \( p_i \) receives a checkpoint message \( m \) from \( p_j \) where \( p_j \in C \), \( p_i \) discards \( m \) because \( m.c_l < r.c_l \). Thus, \( p_i \) can be included in \( C \) at most \( |N^i| \) times where \( |N^i| \) is the number of the neighbor processes of \( p_i \). Therefore, \( |C| \) is incremented by one at most \( I = \sum |N^i| \) times. Since the number of processes in the system is finite, \( I \) is also finite. On the other hand, \( |C| \) is decremented by one if a process \( p_k \notin C \) receives a rollback request message. In our algorithm, once the rollback recovery is invoked by a process in \( C \), rollback request messages are transmitted among the processes in \( C \) while \( C \neq \emptyset \). Therefore, eventually \( |C| = 0 \) and the rollback algorithm is terminated. \( \square \)

Thus, there occurs no livelock in the rollback recovery algorithm.

Theorem 3 The number of rollback request messages transmitted in a rollback class consisting of \( I \) channels is \( O(I) \).

Proof As shown in the proof of the theorem 1, each process \( p_i \) does not receive a checkpoint message more than once from the same neighbor process \( p_j \) included in a rollback class \( C \). Thus, since the rollback is initiated in \( C \) until \( C \) becomes empty, the rollback request is transmitted only once through each channel in the system. Therefore, the number of request messages transmitted in the rollback algorithm is \( O(I) \). \( \square \)

Next, we would like to evaluate the algorithm by comparing with the conventional one \([7]\). Table 1 represents the overhead for checking and rollback recovery. The number of additional messages in \([7]\) is \( O(N) \) where \( N \) is the number of processes in the system and the required time is \( O(D) \) where \( D \) is the diameter of the system because the two-phase commit protocol is used. In our algorithm, no additional message is transmitted for checking and \( O(n) \) additional messages are transmitted and \( O(d) \) time overhead is required for rollback recovery where \( n \) is the number of processes in a rollback class and \( d \) is the diameter of the rollback class. Therefore, our algorithm reduces the number of messages especially in a large-scale distributed system because \( n \ll N \) and \( d \ll D \) are satisfied.

![Table 1: Overhead](attachment:image)

In a large-scale distributed system, most of the elements in \( m.c_l \) are unknown, i.e., \( m.c_l = \bot \) for most of \( k \). Thus, when the algorithm is implemented in a real system, each message \( m \) had better contain only pairs \((k, m.c_l)\) where \( m.c_l \neq \bot \) than the \( n \)-dimensional vector.

7 Concluding Remarks

This paper has proposed the new algorithm for taking checkpoints and rolling back processes in asynchronous distributed systems. The minimum number of processes take checkpoints. The processes are rolled back asynchronously. Each process manipulates \( O(n) \) information and each message contains \( O(n) \) information for the algorithm. The rollback algorithm is terminated with \( O(l) \) message transmissions where \( l \) is
the number of channels. The algorithm realizes the more highly available system than the conventional one because the processes in the system can take the checkpoints without transmitting additional messages and can be asynchronously rolled back without stopping the data transmission. Therefore, the algorithm will play an important role to develop the reliable and available large-scale distributed systems.

References


