Regular Paper

Checkpointing and Restarting Protocols on Object-based Systems

Katsuya Tanaka† and Makoto Takizawa†

In object-based systems, multiple objects cooperate with each other by exchanging messages. The objects may suffer from faults. If some object \( o \) is faulty, \( o \) is rolled back to the checkpoint \( c \) and objects which have received messages from \( o \) are also required to be rolled back to the checkpoints which is consistent with \( c \). In this paper, we discuss how to take checkpoints in object-based systems. Object-based checkpoints are consistent in the object-based system but may be inconsistent according to the traditional message-based definition. We present a protocol for taking object-based checkpoints among objects. An object to take a checkpoint in the traditional message-based protocol does not take a checkpoint if the current checkpoint is object-based consistent with the other objects. The number of checkpoints can be reduced by the object-based protocol.

1. Introduction

Distributed applications are composed of multiple objects. An object is an encapsulation of data and methods for manipulating the data. A method is invoked by a message passing mechanism. On receipt of a request a message with a method \( op \), \( op \) is performed on an object and a response message with the result of \( op \) is sent back. The method may invoke methods on other objects, i.e., invocation is assumed to be nested. A conflicting relation among the methods is defined based on the semantics of the object \(^4\). If a pair of methods \( op_1 \) and \( op_2 \) conflict, a state of the object obtained by performing \( op_1 \) and \( op_2 \) depends on the computation order of \( op_1 \) and \( op_2 \).

In order to increase the reliability and availability, an object takes a checkpoint where a state of an object is saved in the log at a checkpoint. A faulty object \( o \) is rolled back to the checkpoint and then is restarted. Here, objects which have received messages sent by objects rolled back also have to be rolled back. Papers \(^1\),\(^2\),\(^7\),\(^9\),\(^10\),\(^11\),\(^13\) discuss how to take a globally consistent checkpoint of multiple objects.

The paper \(^7\) presents synchronous protocols for taking checkpoints and rolling back objects. The paper \(^9\) presents the concept of significant requests, i.e., the state of an object is changed by performing the request. If an object \( o \) is rolled back, only objects which have received significant requests sent by \( o \) are required to be rolled back. Thus, the number of objects to be rolled back can be reduced. However, in the object-based systems, types of messages, i.e., request and response messages are exchanged among objects and methods are invoked in various ways. In the paper \(^9\), the transmissions of requests and responses and types of invocations are not considered. Since the traditional checkpoints are defined in terms of messages exchanged among objects, the definition is referred to as message-based.

We newly define object-based consistent (O-consistent) checkpoints which can be taken based on conflicting relations among methods in various types of invocations like synchronous and asynchronous ones. The O-consistent checkpoint may be inconsistent with the traditional message-based definition. In this paper, we present a protocol where O-consistent checkpoints are taken for objects without suspending the computation of methods. By taking only the O-consistent checkpoints, the number of checkpoints can be reduced.

In Section 2, we discuss the object-based checkpoints. In Sections 3 and 4, we show a checkpointing protocol and restarting protocol, respectively.

2. Object-based Checkpoints

In this section, we formalize a concept of objects, especially define a conflicting relation
among methods. Then, based on conflicting relation, we discuss what types of checkpoints can be consistently taken in object-based systems.

2.1 Objects

A distributed system is composed of multiple objects \( o_1, \ldots, o_n \). Each object \( o_i \) is an encapsulation of data and a set of methods for manipulating the data. In this paper, we assume methods are synchronously or asynchronously invoked by using the remote procedure call. On receipt of a request \( op \), \( op \) is performed on the object \( o_i \). Here, let \( op^j \) denote an instance of \( op \), i.e., a thread of \( op \) on \( o_i \). Then, a response message is sent back. \( op \) may furthermore invoke another method \( op_1 \), i.e., invocation is assumed to be nested. If \( op_1 \) is synchronously invoked, \( op \) blocks until the response of \( op_1 \). In the asynchronously invocation, \( op \) is being performed without blocking. It is defined that a message \( m \) participates in a method \( op \) if \( m \) is a request or response of \( op \). Let \( Op(m) \) denote a method in which a message \( m \) participates.

Let \( op(s) \) denote a state obtained by performing a method \( op \) on a state \( s \) of an object \( o_i \). \( op_1 \cdot op_2 \) shows that a method \( op_2 \) is performed after \( op_1 \) completes. \( op_1 \) and \( op_2 \) of an object \( o \) are defined to be compatible iff \( op_1 \cdot op_2(s) \) is equivalent with \( op_2 \cdot op_1(s) \) for every state \( s \) of \( o \). Otherwise, \( op_1 \) and \( op_2 \) conflict. It is assumed that an object supports two kinds of methods, i.e., update method which changes the state of the object and non-update one. The types of methods are assumed to be specified with the conflicting relation among the methods in the definition of the object.

2.2 Object-based Checkpoints

A local checkpoint \( c^i \) for an object \( o_i \) is taken where a state of \( o_i \) is stored in the log \( l_i \). If \( o_i \) is faulty, \( o_i \) is rolled back to \( c^i \) by restoring the state stored in the log \( l_i \). Then, other objects have to be rolled back to the checkpoints if they had received messages sent by \( o_i \). A global checkpoint \( c \) is defined to be a tuple \( \langle c^1, \ldots, c^n \rangle \) of the local checkpoints. From here, a term checkpoint means a global one.

Suppose an instance \( op_1^j \) invokes a method \( op_2 \) in \( o_j \). Figure 1 shows possible checkpoints for \( o_i \) and \( o_j \). Here, \( c_3 \) is not taken if \( op_2^j \) is synchronously invoked. Let \( \pi_j(op^j, c^j) \) be a set of instances performed on \( o_j \), which precede \( op^j \) and succeed \( c^j \) or are being performed at \( c^j \) in \( o_j \). For example, \( \pi_j(op_2^j, c_1^j) \) is \( \{op_2^{j_1}, \ldots, op_2^{j_k}\} \) in Fig. 1.

We discuss whether or not each checkpoint \( \langle c_1^k, c_2^k \rangle \) can be taken in the object-based system. For example, \( \langle c_1^j, c_3^j \rangle \) is message-based inconsistent in Fig. 1 because a message \( m_1 \) is an orphan. If \( op_2^j \) is non-update, the state denoted by \( c_2^j \) is the same as \( c_3^j \) and \( c_4^j \). That is, \( \langle c_1^j, c_3^j \rangle \) and \( \langle c_1^j, c_4^j \rangle \) show the same state as \( \langle c_1^j, c_2^j \rangle \). \( \langle c_1^j, c_2^j \rangle \) is message-based consistent. Hence, \( o_j \) can be restarted from any of \( c_3^j \) and \( c_4^j \) if \( o_j \) can be restarted from \( c_2^j \). Here, \( \langle c_1^j, c_3^j \rangle \) is consistent in the object-based system (O-consistent). \( \langle c_1^j, c_4^j \rangle \) is also O-consistent. A local checkpoint \( c^j \) is defined to be complete if there is no method being performed at \( c^j \). For example, \( c_3^j \) is incomplete in Fig. 1. Table 1 summarizes the message-based inconsistent but O-consistent checkpoints, where checkpoints marked * are incomplete if \( op_2^j \) is being performed.

[Definition] A message \( m \) is influential iff a
method instance $op^j$ of an object $o_i$ sends a message $m$ to $o_i$ and one of the following conditions is satisfied:

1. $op^j_1$ is an update type if $m$ is a request message, i.e., $op^j_1$ invokes $op^j_1$ in $o_i$.
2. If $m$ is a response of $op^j_2$, $op^j_2$ is an update type or conflicts with some instance in $\pi_j(op^j_2, c)$ where $c$ is a local checkpoint most recently taken in $o_j$.

If $op^j$ is aborted, only instances receiving influential messages from $op^j$ are required to be aborted. In Fig. 1, suppose $op^j_1$ sends an asynchronous update request $m_1$. Here, $m_1$ is influential from the definition. If $o_i$ is rolled back to $c^i_2$, $o_j$ is also rolled back.

**Definition** A global checkpoint $c (= (c^1, \ldots, c^n))$ is object-based consistent (O-consistent) iff there is no influential orphan message at $c$. □

### 3. Checkpointing Protocol

In this section, a communication-induced protocol used for taking O-consistent checkpoints is introduced. By the protocol, consistent global checkpoint can be taken without suspending the computation. First, a basic communication-induced protocol is presented. Then, we discuss how to take only O-consistent checkpoints in the protocol. Finally, we show how to solve the problems which may be occurred while taking the checkpoints, i.e., cyclic checkpointing and cascading rollback.

#### 3.1 Communication-induced Protocol

We briefly present a basic communication-induced checkpointing protocol where objects are not suspended while checkpoints are being taken. First, each object $o_i$ initially takes a local checkpoint $c^i_0$. An initial checkpoint $⟨c^i_0, \ldots, c^i_0⟩$ is assumed to be consistent. After sending and receiving messages, a first local checkpoint $c^i_1$ is taken for $o_i$. Thus, the $t$-th local checkpoint $c^i_t$ is taken after $c^i_{t-1}$ ($t > 0$). Here, $t$ is defined to be a checkpoint identifier of $c^i_t$.

Suppose a local checkpoint $c^i_t$ is taken for an object $o_i$ after $c^i_{t-1}$. Then, only if $o_i$ sends a message $m$ to another object $o_j$, $m$ is marked checkpointed. By sending $m$, $o_i$ notifies the destination objects that $o_i$ has taken $c^i_t$. Thus, $o_i$ does not send any additional control message to take local checkpoints. Here, suppose $c^i_{u-1}$ is taken for $o_j$ and a checkpoint $⟨c^i_{u-1}, c^j_{u-1}⟩$ is consistent. On receipt of the checkpointed message $m$ from $o_i$, a local checkpoint $c^j_u$ is taken for $o_j$ at which $o_j$ saves a state which is most recent before $o_j$ receives $m$. The state saved here is referred to as checkpoint state. In fact, a current state and the operation $rec(m)$ for receiving $m$ are stored in the log $l_j$. A compensating operation $\sim rec(m)$ to remove every effect done by $rec(m)$ is assumed to be supported for every object. If $o_j$ is rolled back to $c^j_u$, the state saved in the log is first restored, and then $\sim rec(m)$ is performed (Fig. 2).

In the object-based system, $o_j$ does not take $c^j_u$ if $⟨c^i_t, c^j_{u-1}⟩$ is O-consistent. We discuss how $o_j$ decides if $⟨c^i_t, c^j_{u-1}⟩$ is O-consistent.

#### 3.2 O-consistent Checkpoints

A vector of checkpoint identifiers $⟨cp_1, \ldots, cp_n⟩$ is manipulated for an object $o_i$ to identify the $t$-th local checkpoint $c^i_t$ of $o_i$. Each $cp_k$ is initially 0. Each time a local checkpoint is taken for $o_i$, $cp_i$ is increased by one. A message $m$ which $o_i$ sends to $o_j$ after taking $c^i_{cp}$ carries a vector $m.cp$ which is equal to $⟨m.cp_1, \ldots, m.cp_n⟩$, where $m.cp_k$ is $cp_k$ of $o_i$ ($k = 1, \ldots, n$).

On receipt of a message $m$ from $o_j$, the value of $m.cp_j$ is stored in $cp_j$ of $o_i$. $cp_i$ shows a checkpoint identifier which $o_i$ has most recently taken. Another variable $cp_h$ shows a newest checkpoint identifier of an object $o_h$ which $o_i$ knows ($h = 1, \ldots, n, j \neq i$). That is, $⟨cp_1, \ldots, cp_h⟩$ shows a current checkpoint which $o_i$ knows. If $m.cp_j > cp_j$ in $o_i$, $o_i$ finds that $o_j$ has taken $c^j_u$ following $c^j_{cp_j}$ where $u$ is equal to $m.cp_j$. A local checkpoint $c^i_t$ is identified by a vector $⟨c^i_t.cp_1, \ldots, c^i_t.cp_n⟩$ where each $c^i_t.cp_j$ shows a value of $cp_j$ when $c^i_t$ is taken for.

![Fig. 2 Checkpoint state.](image)
o_j.

A local checkpoint c_i has a bitmap c_i.BM which is equal to b_1 \cdots b_n where each h-th bit b_h is used for an object o_h (h = 1, \ldots, n). Suppose c_i is taken for o_i. Here, c_i.b_i is 1 and c_i.b_j is 0 for j = 1, \ldots, n, j \neq i. If c_i.b_j is 0 and there is data to be sent to o_j, o_i sends a checkpointed message m with the data to o_j. Here, the value of c_i.BM is stored in m.BM.

On receipt of m from o_i, o_j takes a local checkpoint c_j. Here, the value of m.b_k is stored in c_j.b_k (k = 1, \ldots, n, k \neq j) and c_j.b_j is updated to 1 while the checkpoint identifier vector is updated as presented here. Thus, “c_i.b_k = 1” shows that o_i knows o_k takes a local checkpoint by the checkpointing protocol initiated by a same object.

**[Definition]** c_i and c_j are in the same generation if c_i.BM \cap c_j.BM \neq \phi and c_i.cp_k is equal to c_j.cp_k for every object o_k such that c_i.b_k = c_j.b_k = 1.

Each time an object o_i sends a message m to o_j, a message sequence number sq and a subsequence number ssq_i are incremented by one (j = 1, \ldots, n). The sequence number m.sq and a vector of the subsequence numbers m.ssq (= \langle m.ssq_1, \ldots, m.ssq_n \rangle) are carried by m. Variables rssq_1, \ldots, rssq_o and rssq_1, \ldots, rssq_o are manipulated in o_j. On receipt of m from o_i, o_j accepts m if m.ssq_i is equal to rssq_i + 1. That is, o_j delivers messages from each object in the sending order. Then, rssq_i is incremented by one and the value of m.sq is stored in rssq_i. rssq_i, and rssq_j show subsequence and sequence numbers of message which o_j has most recently received from o_i. m also carries a vector m rq (= \langle m.rq_1, \ldots, m.rq_n \rangle) where m.rq_k is equal to rssq_k (k = 1, \ldots, n). Here, m.rq_k shows a sequence number of message which o_i has received from o_j just before c_i and t is equal to m.cp_i (k = 1, \ldots, n).

On receipt of a message m from o_i, o_j collects a set M_j of messages m_{j1}, \ldots, m_{jl_j} which o_j has sent to o_i after c_{i-1} and o_i has received before c_i. Here, m_{j1}.sq \leq m.rq_j (Fig. 3). Messages which o_j sends after c_{i-1} are stored in the sending log of o_j. Suppose o_j receives a checkpointed message m from o_i. If m.cp_i > cp_i, o_j knows o_i takes c_i. o_j collects every message m' which o_j has sent after c_{i-1} and m'.sq < m.rq_j in the set M_j.

It is clear for the following theorem to hold from the definition.

**[Theorem]** A message m_{j1} which o_j sends to o_i after taking a local checkpoint c_{i-1} before c_i is influential if m_{j1} is a request and Op(m_{j1}) is an update type, or m_{j1} is a reply and Op(m_{j1}) is an update type or conflicts with some update method in \pi_j(Op(m_{j1}), c_{i-1}).

The condition of the theorem is referred to as influential message (IM) condition. Only if some message in M_j is decided to be influential by IM condition, o_j takes a local checkpoint.

### 3.3 Cyclic Checkpointing

We discuss how to resolve a cyclic checkpointing which occurred in the communication-induced protocol. Due to the cyclic checkpointing, the checkpointing procedure cannot be terminated as shown in Example 1.

**[Example 1]** Suppose each of three objects o_1, o_2, and o_3 has initially checkpoint identifier vector cp = \langle cp_1, cp_2, cp_3 \rangle = \langle 4, 2, 7 \rangle (Fig. 4). First, a local checkpoint c_5 is taken for o_1. Here, cp is \langle 5, 2, 7 \rangle. o_1 sends m_1 with \langle 5, 2, 7 \rangle to o_2 after taking c_5. o_2 takes c_5 on receipt of m_1
where $c^2_Bm$ is $\langle 5, 3, 7 \rangle$. Then, $o_2$ sends $m_2$ with $\langle 5, 3, 7 \rangle$ to $o_3$. On receipt of $m_2$, $o_3$ takes $c^2_B$ and sends $m_3$ with $\langle 5, 3, 8 \rangle$ to $o_1$. $o_1$ takes $c^2_B$. Then, $o_2$ and $o_3$ take new local checkpoints as presented here. Thus, the checkpointing procedure cannot be terminated in $o_1$, $o_2$, and $o_3$. This is cyclic checkpointing.

Here, when $o_1$ receives $m_3$, $o_1$ is not required to take a local checkpoint because a checkpoint $\langle c^1_B, c^3_B, c^3_B \rangle$ taken already is consistent. A pair of checkpoints identified by $\langle 5, 2, 7 \rangle$ and $\langle 5, 3, 8 \rangle$ are in the same generation.

The cyclic checkpointing is resolved by using the bitmap $BM$ as shown in Example 2.

**Example 2** Here, let a notation $\langle cp_1, \ldots, cp_n \rangle_{b_1 \ldots b_n}$ show $cp$ is $\langle cp_1, \ldots, cp_n \rangle$ and $BM$ is $b_1 \ldots b_n$. In Fig. 4, $o_1$ sends $o_2$ a message $m_1$ with $\langle 5, 2, 7 \rangle_{100}$, i.e., $cp = \langle 5, 2, 7 \rangle$ and $BM = 100$ after $c^1_B$. On receipt of $m_1$, $cp$ is changed to $\langle 5, 2, 7 \rangle$ in $o_2$. Then, $o_2$ sends $m_2$ with $\langle 5, 3, 7 \rangle_{110}$ to $o_3$ after $c^2_B$. $c^2_B$ is taken for $o_3$ and then sends $m_3$ with $\langle 5, 3, 8 \rangle_{111}$ to $o_1$. On receipt of $m_3$, $o_1$ knows the checkpointing procedure has been initiated by $o_1$ because $\langle 5, 2, 7 \rangle$ and $\langle 5, 3, 8 \rangle$ are in the same generation.

The checkpoint identifier vector $cp$ ($= \langle cp_1, \ldots, cp_n \rangle$) and the bitmap $BM = b_1 \ldots b_n$ are manipulated in $o_1$ on receipt of $m$ as follows:

- $cp_k := \max(cp_k, m.cp_k)$ if $m.b_k = 1$ for every $k (\neq i)$.
- $BM := BM \cup m.BM$.

The checkpoint identifier vector $cp$ and the bitmap $BM$ are saved in the checkpoint log $c^i_Bm$ of $o_i$ only if they are changed. In Fig. 4, on receipt of $m_3$, $c^3_Bm$ is updated to $\langle 5, 3, 8 \rangle$. If $cp_1 > m.cp_1$, another object initiates the checkpointing procedure independently of $o_1$. A local checkpoint is taken for $o_1$ if there is some influential message in $M_1$.

3.4 Merge of Checkpoints

Next, we consider a cascading rollback problem which occurred while rolling back the objects as shown in the following example.

**Example 3** In Fig. 5, every object has a checkpoint identifier vector $cp$ which is equal to $\langle 4, 3, 7, 2 \rangle$. Suppose $o_1$ and $o_4$ independently take checkpoints. $o_1$ sends $m_1$ after $c^3_B$ with $\langle 5, 3, 7, 1 \rangle_{1000}$, i.e., $cp$ is $\langle 5, 3, 7, 1 \rangle$ and $BM$ is 1000. On receipt of $m_1$, $o_2$ takes $c^2_B$ and then sends $m_2$ with $\langle 5, 4, 7, 1 \rangle_{1100}$. On the other hand, $o_4$ takes $c^2_B$ with $\langle 4, 3, 7, 2 \rangle_{0001}$ and then sends $m_4$ to $o_3$. $o_3$ takes $c^3_B$ with $\langle 4, 3, 8, 2 \rangle_{0011}$ and then sends $m_3$ to $o_2$. $o_2$ receives $m_3$ with $\langle 4, 3, 8, 2 \rangle_{0011}$ from $o_3$ after $c^2_B$ with $cp$ which is equal to $\langle 5, 4, 7, 1 \rangle$. $o_3$ receives $m_2$ with $\langle 5, 4, 7, 1 \rangle_{1100}$ after $c^3_B$ with $cp$ which is equal to $\langle 4, 3, 8, 2 \rangle$. One way is that $o_2$ and $o_3$ take $c^3_B$ with $\langle 5, 4, 7, 1 \rangle_{1100}$ and $c^3_B$ with $\langle 5, 4, 9, 3 \rangle_{1110}$, respectively. Here, $\langle c^3_B, c^3_B, c^3_B, c^3_B \rangle$ and $\langle c^3_B, c^3_B, c^3_B, c^3_B \rangle$ are taken for $o_1$, $o_2$, $o_3$, and $o_4$.

Suppose $o_4$ is faulty and is rolled back to $c^3_B$. Then, $o_3$ is rolled back to $c^3_B$ and then $o_2$ is rolled back to $c^3_B$. Here, $o_3$ is required to be furthermore rolled back to $c^3_B$ and $o_3$ is also rolled back to $c^3_B$. In the worst case, each object is rolled back to the local checkpoints $n$ times for the number $n$ of objects.

In order to prevent such a cascading rollback, we take an approach to merging multiple checkpoints to one. In Fig. 5, $o_2$ receives $m_3$ after $c^3_B$. Here, $\langle c^3_B, c^3_B \rangle$ with $BM$ which is equal to 1100 and $\langle c^3_B, c^3_B \rangle$ with $BM$ which is equal to 0011 are merged into $\langle c^3_B, c^3_B, c^3_B, c^3_B \rangle$ with $BM$ which is equal to 1111.

**[Merge of checkpoints]** After $c^i_B$, $o_i$ receives a message $m$.

1. If a checkpoint $c^i_B$ denoted by $m.cp$ is not in the same generation as $c^j_B$, i.e., $c^i_B.BM \cap m.BM$ is not $\phi$, the value of $m.cp$ is stored in $c^i_B.cp_k$ if $c^j_B.b_k = 0$ and $m.b_k = 1$ for every $k (\neq i)$, and $c^i_B.BM$ is updated to $c^j_B.BM \cup m.BM$.

2. Otherwise, $c^i_B.BM$ is updated to $c^j_B.BM \cup m.BM$ and $c^i_B.cp_k$ is changed to max($c^i_B.cp_k, m.cp_k$) for every $k (\neq i)$.

**[Theorem]** A set of local checkpoints which belong to the same generation with the merge
procedure are O-consistent. □

[Proof] We prove the theorem by contradiction. Assume there are a pair of local checkpoints $c^j_i$ and $c^u_i$ of the same generation, which are not O-consistent, i.e., there exists an influential message $m$ which is sent after $c^j_i$ and is received before $c^j_i$.

Here, if $o_i$ sends $m$ to $o_j$, $m$ is marked checkpointed. If received message after $m$, $o_j$ takes a local checkpoint $c^u_{i-1}$ most recent before receiving $m$ if $m$ is influential. Otherwise, $o_j$ does not take a local checkpoint. Thus, a pair of the local checkpoints $c^j_i$ and $c^u_i$ never belong to a same generation. This contradicts the assumption. □

By the merging procedure, a new local checkpoint is not taken for $o_2$ even if $o_2$ receives messages after $m_3$ in Fig. 5.

4. Rollback Recovery

If some object is faulty, objects which have received influential messages sent by the object are also required to be rolled back. In this section, we discuss how to restart the computation after some faulty object is rolled back.

4.1 Restarting Protocol

If an object $o_i$ is faulty, $o_i$ is rolled back to the local checkpoint $c^j_i$. Other objects which have received influential messages sent by $o_i$ after $c^j_i$ are also required to be rolled back. Messages which $o_i$ sends are recorded in the sending log. $o_i$ has to send a rollback request message $R$-Req to every object $o_j$ which $o_i$ has sent influential messages after $c^j_i$. In order to decide to which objects $R$-Req is sent, $o_i$ manipulates a log $SL^j_i$ as follows:

- When a local checkpoint $c^j_i$ is taken for $o_i$, $SL^j_i$ is initiated to be empty.
- If $o_i$ sends an influential message $m$ to $o_j$, $SL^j_i$ is updated to $SL^j_i \cup \{o_j\}$.

If $o_i$ is rolled back to $c^u_i$, $o_i$ sends $R$-Req to every object $o_j$ in $SL^j_i$. Here, $R$-Req contains the following information:

- A vector $cp = \{cp_1, \ldots, cp_n\}$ of $c^j_i$ to which $o_i$ is rolled back.
- A bitmap $RB = rb_1 \ldots rb_n$ where each $rb_k$ is 1 if $o_i$ knows $o_k$ is rolled back to a same generation checkpoint as $c^j_i$, otherwise, $rb_k$ is 0.

Suppose an object $o_i$ is faulty and is rolled back to $c^j_i$. $o_i$ sends $R$-Req to every $o_k$ in $SL^j_i$ with $cp$ and $RB$ where $rb_k$ is updated to 1. Then, $o_i$ is suspended. On receipt of $R$-Req from $o_i$, $o_j$ is also suspended. $o_j$ discards $R$-Req if $R$-Req.$rv_j$ is 1 since $o_j$ has been already rolled back in this generation. Otherwise, $rb_j$ is changed to 1 and $RB$ is updated to $RB \lor R$-Req.$RB$. $o_j$ looks for an oldest local checkpoint $c^j_i$ where $cp_i$ is equal to $R$-Req.$cp_i$. If $o_j$ finds $c^j_i$, $c^j_i$ is defined to be a rollback point of $o_j$. Otherwise, the most recent checkpoint where $cp_i$ is smaller than $R$-Req.$cp_i$ is a rollback point.

Then, $o_j$ collects a set $RL^j_i$ of messages which $o_j$ has received after $c^j_i$. If there is some influential message in $RL^j_i$, $o_j$ is rolled back to the rollback point $c^j_i$. Then, $o_j$ sends $R$-Req to every $o_k$ in $SL^j_i$ with $RB$ and $cp$. If $o_j$ receives no influential message from $o_i$, $o_j$ discards $R$-Req since $o_j$ is not required to be rolled back. If $o_j$ does not send $R$-Req to any objects, $o_j$ sends the restart request message $Res$-Req to $o_i$. Otherwise, $o_j$ waits for $Res$-Req from every object in $SL^j_i$. Then, $o_j$ sends $Res$-Req to $o_i$.

[Example 4] In Fig. 6, if $o_1$ is faulty, $o_1$ is rolled back to $c^j_1$. $o_1$ is suspended and finds that $o_2$ has sent an influential message to $o_2$ by searching $SL^j_1$. Then, $o_1$ sends $R$-Req to $o_2$ with $cp = \langle 1, 0, 0 \rangle$ and $RB = 100$. On receipt of $R$-Req from $o_1$, $o_2$ finds an oldest local checkpoint $c^2_1$ where $cp_1$ is equal to 1 because $R$-Req.$cp_1$ is 1. Since $m_1$ in $RL^j_i$ is influential, $o_2$ is rolled back to $c^j_1$. $R$-Req.$rb_2$ is updated to 1. Then, $o_2$ sends $R$-Req to $o_3$ if $m_2$ is influential. On receipt of $R$-Req from $o_2$, $o_3$ is rolled back to $c^2_2$ if $m_2$ is influential. Otherwise, $o_3$ just discards $R$-Req, sends back the $Res$-Req to $o_2$, and then continues the computation. On receipt of $Res$-Req from $o_3$, $o_2$ sends $Res$-Req to $o_1$ and is
restarted.

4.2 Synchronous Restarting Protocol

In the protocol, each object is not required to be restarted simultaneously with other objects. This protocol is effective if only a few number of objects are rolled back after some faulty object is rolled back. However, the more number of objects to be rolled back, the longer it takes to recover from the fault. In order to resolve the difficulty, we show a synchronous restarting protocol.

Suppose an object \( o \) is faulty and is rolled back to the local checkpoint \( c^i_o \). \( o \) is suspended and broadcasts \( R-Req \) to all objects with \( c^i_o.cp \). On receipt of \( R-Req \) from \( o \), \( o_j \) is suspended. Then, the value \( c^i_o.cp \) is compared with \( R-Req.cp_i \) where \( c^i_u \) is a most recent local checkpoint. Suppose \( R-Req.cp_i \geq c^i_u.cp_i \). Since \( o_j \) has not taken a same generation checkpoint with \( c^i_t \), \( o_j \) is not rolled back. \( o_j \) sends back a message \( \text{no} \) to \( o \) and then is restarted. Otherwise, \( o_j \) sends \( \text{yes} \) to \( o \). \( o_i \) finds a group of objects to be rolled back by using a bitmap \( RB \) (= \( rb_1, \ldots, rb_n \)). Each variable \( rb_k \) is initially 0 (1 \( \geq k \geq n \)). On receipt of \( \text{yes} \) from \( o_k \), \( rb_k \) is updated to 1. After receiving messages from all the objects, \( o_i \) sends \( \text{Rollback} \) with \( RB \) to \( o_k \) where \( rb_k \) is 1. On receipt of \( \text{Rollback} \) from \( o_i \), \( o_j \) is rolled back to the rollback point if \( o_j \) had received any influential message from \( o_k \) where \( rb_k \) is 1. Then, \( o_j \) sends \( \text{Done} \) to \( o \). On receipt of \( \text{Done} \) from all the objects which \( o_j \) has sent \( \text{Rollback} \), \( o_i \) sends \( \text{Res-Req} \) to the objects and then is restarted. On receipt of \( \text{Res-Req} \), \( o_j \) is restarted.

**Example 5** Suppose there are four objects \( o_1, o_2, o_3 \) and \( o_4 \) as shown in Fig. 7. Here, suppose \( o_4 \) is faulty and is rolled back to the checkpoint \( c^{14}_o \). \( o_1 \) broadcasts \( R-Req \). On receipt of \( R-Req \) from \( o_1, o_2 \) and \( o_3 \) send \( \text{yes} \) and \( o_4 \) sends \( \text{no} \) to \( o_1 \) since \( c^{14}_o.cp_1 = c^{2}_o.cp_1 = c^{3}_o.cp_1 \). On receipt of the messages, \( rb \) is updated to 1110 in \( o_1 \). \( o_1 \) sends \( \text{Rollback} \) to \( o_2 \) and \( o_3 \). On receipt of \( \text{Rollback} \), \( o_2 \) is rolled back to \( c^2_o \) if \( m_1 \) is influential. Similarly, \( o_3 \) is rolled back to \( c^3_o \) if \( m_2 \) is influential.

5. Concluding Remarks

We discussed how to take object-based consistent (\( O \)-consistent) checkpoints which show consistent global checkpoints in object-based systems. \( O \)-consistent checkpoints may be inconsistent with the traditional message-based definition. We have defined influential messages on the basis of the conflicting relation of requests where the methods are synchronously or asynchronously invoked in the nested manner. Only objects receiving influential messages are rolled back if the senders of the influential messages are rolled back. As a result, the number of local checkpoints can be reduced by the \( O \)-checkpoints. At the \( O \)-consistent checkpoint, there is no orphan influential message. Also, we presented the protocol for taking \( O \)-consistent checkpoints where no object is suspended in taking checkpoints. We presented the restarting protocol after some faulty object is rolled back.

**References**


Checkpointing and Restarting Protocols on Object-based Systems

Katsuya Tanaka was born in 1971. He received his B.E. and M.E. degrees in Computers and Systems Engineering from Tokyo Denki University, Japan in 1995 and 1997, respectively. From 1997 to 1999, he worked for NTT Data Corporation. Currently, he is a assistant in the Department of Computers and Systems Engineering, Tokyo Denki University. He received the D.E. degree from Dept. of Computers and Systems Engineering, Tokyo Denki University, Japan, in 2000. His research interest includes distributed systems, transaction management, recovery protocols, and network protocols. He is a member of IEEE CS and IPSJ.

Makoto Takizawa was born in 1950. He received his B.E. and M.E. degrees in Applied Physics from Tohoku Univ., Japan, in 1973 and 1975, respectively. He received his D.E. in Computer Science from Tohoku Univ. in 1983. From 1975 to 1986, he worked for Japan Information Processing Developing Center (JIPDEC) supported by the MITI. He is currently a Professor of the Dept. of Computers and Systems Engineering, Tokyo Denki Univ. since 1986. From 1989 to 1990, he was a visiting professor of the GMD-IPSI, Germany. He is also a regular visiting professor of Keele univ., England since 1990. He was a program co-char of IEEE ICDCS-18, 1998 and serves on the program committees of many international conferences. He chaired SIGDPS of IPSJ from 1997 to 1999. He is IPSJ fellow. His research interest includes communication protocols, group communication, distributed database systems, transaction management, and security. He is a member of IEEE, ACM, and IPSJ.