マルチメディア通信と分散処理 98－2
(2000. 6. 8)

Checkpointing Protocol for Object-Based Systems

Katsuya Tanaka and Makoto Takizawa

Tokyo Denki University
Email {katsu, taki}@takilab.k.dendai.ac.jp

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.

分散オブジェクト環境におけるロールバック復旧方式と評価

田中 勝也　滝沢 誠

東京電機大学

E-mail {katsu, taki}@takilab.k.dendai.ac.jp

分散オブジェクト環境では、通信網で相互接続された複数のオブジェクトがメッセージの送受信により協調動作を行う。あるオブジェクトからの要求メッセージの受信において、オブジェクトは要求されたメソッドを起動し、応答メッセージを返す。さらに、起動されたメソッドが他のオブジェクトのメソッドを起動する場合もある。本論文では、分散オブジェクトシステムにおける意味的に正しい状態を定義し、正しい状態でチェックポイントを取得するためのプロトコル、及びロールバックのためのプロトコルを提案する。また、評価により、本手法により、従来の方式よりも取得するチェックポイント数が削減されることを示す。

1 Introduction

Distributed applications are composed of multiple objects cooperating by exchanging messages through networks. 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 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 *op* may invoke methods on other objects, i.e. invocation is *nested*. A conflicting relation among the methods is defined based on the semantics of the object [3]. 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 the state is saved in the *log*. A faulty object *o* is *rolled back* to the checkpoint and then the computation is restarted. Here, objects which have received messages sent by the object rolled back also have to be rolled back so that there is no *orphan* message [2], i.e. messages sent by no object but received by some object.

Papers [1, 2, 4–7, 10] discuss how to take a globally consistent checkpoint for multiple objects. The paper [4] presents synchronous protocols for taking checkpoints and rolling back objects. The paper [5] presents the concept of *significant* requests, i.e. the state of an object is changed by performing the request. If the 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, different types of messages, i.e. *request* and *response* messages are exchanged among the objects and methods are invoked in various ways. In the paper [5], the transmissions of requests and responses and types of invocations are not considered. Since the traditional consistent checkpoints are defined in terms of messages exchanged among objects, the definition is referred to as *message-based*.

We 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 in object-based systems. The O-consistent checkpoint may be inconsistent with the traditional message-based definition. In this paper, we present a communication-induced 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 taken by objects can be reduced.

In section 2, we discuss the object-based checkpoints. In section 3, we show a protocol for taking O-consistent checkpoints. In section 4, we present how to restart the objects. In section 5, we evaluate the protocol by comparing with the message-based protocol.

2 Object-Based Checkpoints

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 collection 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* message *m* with a method *op*, *op* is performed on the object o_i. Here, let op^i denote an instance of *op*, i.e. a thread of *op* on o_i. Then, the *response* message with the result of *op* is sent back. The method *op* may furthermore invoke another method op_1, i.e. invocation of *op* is *nested*. If op_1 is synchronously invoked, *op* blocks until receiving the response of op_1. In the asynchronously invocation, op eventually receives the response of op_1 but op is being performed without blocking.

Let $op(s)$ denote a state obtained by performing a method op on a state s of an object o_i. $op_1 \circ op_2$ shows that a method op_2 is performed after op_1 completes. A pair of methods op_1 and op_2 of an object o are *compatible* iff $op_1 \circ op_2(s) = op_2 \circ op_1(s)$ for every state s of o_i [3]. op_1 and op_2 *conflict* iff they are not compatible.

Each method op is performed on an object o_i in an atomic manner. Only if op commits, the change of o_i done by op can be viewed by other methods. Each object supports some synchronization mechanism like locking to realize the atomicity.

An object supports two kinds of methods, i.e. *update* method which changes the state of the object and *non-update* one which does not change the state. For example, *deposit* of a *Bank* object is an update method and *check* is a non-update one.

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.

2.2 Object-based checkpoints

An object o_i takes a local checkpoint c^i where the state of o_i is stored in the log l_i ($i = 1, \ldots, n$). If the object o_i is faulty, o_i is rolled back to the local checkpoint 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 the object o_i. A *global checkpoint* c is 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^i of an object o_i invokes a method op_2 in another object o_j. Figure 1 shows possible checkpoints to be taken in the objects o_i and o_j. Here, no local checkpoint c_3^i is taken if op_2^j is synchronously invoked because op_1^i blocks after invoking op_2^j. Let $\pi_j(op^j, c^j)$ be a set of instances performed on an object o_j for a local checkpoint c^j, which

- precede op^j and
- succeed a local checkpoint c^j or are being performed at c^j in o_j.

For example, $\pi_j(op_2^j, c_1^j) = \{op_{21}^j, \ldots, op_{2l}^j\}$ in Figure 1.

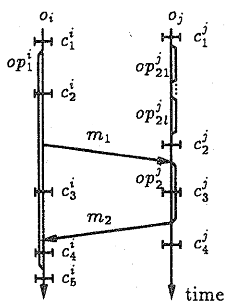

Figure 1: Possible checkpoints.

A local checkpoint c^i is *complete* if there is no method being performed at c^i. For example, c_3^i is incomplete in Figure 1. Suppose the object o_j is rolled back to a local checkpoint c_h^j. If c_h^j is complete, the state of o_j is just restored from the log. If c_h^j is incomplete, every method being performed at c_h^j has to be aborted after the state is restored. However, no method invoked by a non-update method is required to be rolled back.

Table ?? summarizes the message-based inconsistent but O-consistent checkpoints in Figure 1, where checkpoints marked * are incomplete if op_2^j is being performed. For example, a checkpoint $\langle c_1^i, c_4^j \rangle$ is O-consistent if op_2^j is a non-update method.

If another object o_j took a local checkpoint, o_i has to decide whether or not to take a new local checkpoint. We define an *influential* message such that o_i takes a local checkpoint only if o_i receives influential messages from o_j.

[Definition] A message m is *influential* iff a method instance op_2^j of an object o_j sends a message m to another object o_i and one of the following conditions is satisfied:

1. op_1^i is an update type if m is a request message, i.e. op_2^j invokes op_1^i in o_i.
2. If m is a response message of op_2^j, op_2^j is an update type or conflicts with some instance in $\pi_j(op_2^j, c)$ where c is a local checkpoint most recently taken in o_j. □

If an instance op^i is aborted, only instances receiving influential messages from op^i are required to be aborted.

Based on the definition of influential messages, O-consistent checkpoints are defined as follows.

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

For detailed discussion about the definition of O-consistent checkpoints, see [8, 9].

3 Checkpointing Protocol

3.1 Communication-induced protocol

We briefly present a basic communication-induced protocol for taking message-based consistent checkpoints among objects where objects are not suspended while checkpoints are being taken. First, each object o_i is assumed to initially take a local checkpoint c_0^i where the initial state of o_i is saved in the log l_i. An initial checkpoint $\langle c_0^1, \ldots, c_0^n \rangle$ is assumed to be consistent. After sending and receiving messages, the object o_i takes a first local checkpoint c_1^i. Thus, o_i takes the tth local checkpoint c_t^i after taking the $(t-1)$th local checkpoint c_{t-1}^i $(t > 0)$. The object o_i sends and receives messages after taking c_{t-1}^i before c_t^i. Here, t denotes a *checkpoint identifier* of c_t^i. The checkpoint identifier is incremented by one each time a local checkpoint is taken.

Suppose o_i autonomously takes a succeeding local checkpoint c_t^i after taking c_{t-1}^i. Then, only if there is a message m which o_i sends to another object o_j, m is marked *checkpointed*. By sending m, the object o_i notifies the destination objects that o_i has taken c_t^i. Thus, o_i does not send any additional control message to require other objects to take local checkpoints. Here, suppose that a local checkpoint c_{u-1}^j is taken in the object o_j and a checkpoint $\langle c_{t-1}^i, c_{u-1}^j \rangle$ is consistent. On receipt of the checkpointed message m from o_i, the object o_j takes a local checkpoint c_u^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 the local checkpoint c_u^j, a following procedure is performed.

1. The state saved in the log is first restored.
2. The compensating operation $\sim rec(m)$ is performed for $rec(m)$ saved in the log.

Here, o_j can be rolled back to a checkpoint state. Each object takes a local checkpoint without stopping the communication.

3.2 O-consistent checkpoints

A vector of checkpoint identifiers $\langle cp_1, \ldots, cp_n \rangle$ is manipulated for an object o_i to identify the tth local checkpoint c_t^i of o_i. Each variable cp_k is initially 0. If the object o_i takes a local checkpoint, the checkpoint identifier cp_i is incremented by one, i.e. $cp_i := cp_i + 1$. A message m which o_i sends to o_j after taking $c_{cp_i}^i$ carries a vector of checkpoint identifiers $m.cp = \langle m.cp_1, \ldots, m.cp_n \rangle$, where $m.cp_k$ is cp_k of o_i ($k = 1, \ldots, n$).

On receipt of a message m from another object o_j, $cp_j := m.cp_j$ in an object o_i. The variable 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, $\langle c_{cp_1}^i, \ldots, c_{cp_n}^i \rangle$ 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 a checkpoint c_u^j following $c_{cp_j}^i$ where $u = m.cp_j$. A local checkpoint c_t^i is identified by a checkpoint identifier vector $\langle c_t^i.cp_1, \cdots, c_t^i.cp_n \rangle$ where each $c_t^i.cp_j$ shows a value of a variable cp_j when c_t^i is taken in o_i.

A local checkpoint c_t^i has a bitmap $c_t^i.BM = b_1 \cdots b_n$ where each hth bit b_h is used for an object o_h ($h = 1, \ldots, n$). Suppose an object o_i initiates a checkpointing procedure after taking c_{t-1}^i and then o_i takes a local checkpoint c_t^i. Here, $c_t^i.b_i = 1$ and $c_t^i.b_j = 0$ for $j = 1, \ldots, n$, $j \neq i$. If $c_t^i.b_j = 0$ and there is data to be sent to another object o_j, o_i sends a checkpointed message m with the data to o_j. Here, $m.BM := c_t^i.BM$.

On receipt of m from o_i, an object o_j takes a local checkpoint c_u^j. Here, $c_u^j.b_k := m.b_k$ (for $k = 1, \ldots, n$, $k \neq j$) and $c_u^j.b_j := 1$ while the checkpoint identifier vector is updated as presented here. Thus, "$c_t^i.b_k = 1$" shows that o_i knows that an object o_k takes a local checkpoint by the checkpointing protocol initiated by a same object.

[Definition] A pair of local checkpoints c_t^i and c_u^j are in the *same generation* if $c_t^i.BM \cap c_u^j.BM \neq \phi$ and $c_t^i.cp_k = c_u^j.cp_k$ for every object o_k such that $c_t^i.b_k = c_u^j.b_k = 1$. □

Since no orphan message is in the same generation checkpoint, the following theorem holds.

[Theorem] A collection of same generation local checkpoints are message-based consistent. □

Each time an object o_i sends a message m, a message *sequence number* sq is incremented by one. In addition, a *subsequence number* ssq_j is incremented by one if m is sent to an object o_j ($j = 1, \ldots, n$). 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 $rsq_1, \ldots, rsq_n$ and $rssq_1, \ldots, rssq_n$ are manipulated in o_j to receive messages in the sending order and without message loss. On receipt of m from o_i, o_j accepts m if $m.ssq_j = rssq_i + 1$. That is, o_j delivers messages from each object in the sending order. Then, $rssq_i := rssq_i + 1$ and $rsq_i := m.sq$. The variables $rssq_i$ and rsq_i show subsequence and sequence numbers of message which o_j has most recently received from o_i. The message m also carries a vector of the receipt sequence numbers $m.rq = \langle m.rq_1, \ldots, m.rq_n \rangle$ where $m.rq_k = rsq_k$ ($k = 1, \ldots, n$). Here, $m.rq_k$ shows a sequence number of message which o_i has received from o_j just before taking the local checkpoint c_t^i and $t = m.cp_i$ ($k = 1, \ldots, n$).

On receipt of a message m from an object o_i, an object o_j collects a set M_j of messages $m_{j1}, \ldots, m_{jl_j}$ which o_j has sent to o_i after taking the current local checkpoint c_{u-1}^j and o_i has received before taking c_t^i. Here, $m_{jh}.sq \leq m.rq_j$ [Figure 2]. Messages which o_j sends after taking c_{u-1}^j 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 that o_i takes a new local checkpoint c_t^i. o_j collects every message m' which o_j has sent after c_{u-1}^j 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_{jh} which o_j sends to o_h after taking a local checkpoint c_{u-1}^j before c_u^j is *influential* if the following condition holds:

1. $Op(m_{jh})$ is an update type if m_{jh} is a request, or
2. $Op(m_{jh})$ is an update type or conflicts with some update method in $\pi_j(Op(m_{jh}), c_{u-1}^j)$ if m_{jh} is a response. □

The condition of the theorem is referred to as *influential message* (*IM*) condition. If some message in M_j is decided to be influential by the *IM* condition, the object o_j takes a local checkpoint c_u^j showing a checkpoint state of o_j. Otherwise, o_j does not take a local checkpoint even if M_j includes an orphan message.

3.3 Cyclic checkpointing

[Example 1] Suppose there are three objects o_1, o_2, and o_3 in each of which a checkpoint identifier

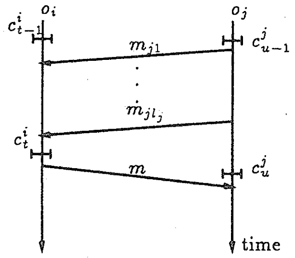

Figure 2: Influential messages.

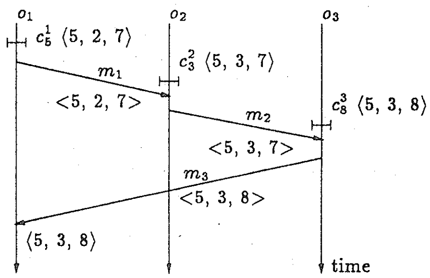

Figure 3: Cyclic checkpointing.

vector is initially ⟨4, 2, 7⟩ [Figure 3]. First, the object o_1 takes a local checkpoint c^1_5. Here, the checkpoint identifier vector is changed to ⟨5, 2, 7⟩. The object o_1 sends a checkpointed message m_1 with ⟨5, 2, 7⟩ to o_2 after taking c^1_5. o_2 takes a local checkpoint c^2_3 on receipt of m_1 where $c^2_3.cp$ = ⟨5, 3, 7⟩. Then, o_2 sends m_2 with ⟨5, 3, 7⟩ to o_3. On receipt of m_2, o_3 takes c^3_8 and sends m_3 with ⟨5, 3, 8⟩ to o_1. o_1 takes c^1_6. 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*. □

In this example, when o_1 receives m_3, o_1 is not required to take a local checkpoint because a checkpoint $\langle c^1_5, c^2_3, c^3_8\rangle$ taken already is consistent. o_1 has to know a pair of checkpoints identified by ⟨5, 2, 7⟩ and ⟨5, 3, 8⟩ are in the same generation. The cyclic checkpointing is resolved by using the bitmap BM as shown in a following example. Here, let a notation "$\langle cp_1, \ldots, cp_n\rangle_{b_1 \ldots b_n}$" show $cp = \langle cp_1, \ldots, cp_n\rangle$ and $BM = b_1 \cdots b_n$.

[Example 2] In Figure 3, the object o_1 sends o_2 a checkpointed message m_1 with $\langle 5, 2, 7\rangle_{100}$, i.e. $cp = \langle 5, 2, 7\rangle$ and $BM = 100$ after taking c^1_5. On receipt of m_1, cp is changed to ⟨5, 2, 7⟩ in o_2. Then, o_2 sends m_2 with $\langle 5, 3, 7\rangle_{110}$ to o_3 after taking c^2_3. On receipt of m_2, o_3 takes a local checkpoint c^3_8 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 checkpoints identified by ⟨5, 2, 7⟩ and ⟨5, 3, 8⟩ are in the same generation. □

On receipt of a checkpointed message m from another object o_j, an object o_i does not take a local checkpoint if $m.cp$ denotes a same generation checkpoint as the local checkpoint $c^i_{cp_i}$ most recently taken by o_i. Hence, the checkpoint identifier vector $cp = \langle cp_1, \ldots, cp_n\rangle$ and the bitmap $BM = b_1 \cdots b_n$ are manipulated in o_i 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 $c^i_{cp_i}$ of o_i only if they are changed. For example, on receipt of the message m_3 from the object o_3, the object o_1 updates cp and BM to be ⟨5, 3, 8⟩ and 111, respectively, in Example 2. Then, o_1 saves cp and BM in the log c^1_5 since they are changed. Here, $c^1_5.cp = \langle 5, 3, 8\rangle$ and $c^1_5.BM = 111$. After receiving m_3, suppose o_1 sends a message m_4 to o_2. m_4 carries $\langle 5, 3, 8\rangle_{111}$. On receipt of m_4, o_2 updates cp and BM. Here, $c^2_3.cp = \langle 5, 3, 8\rangle$ and $c^2_3.BM = 111$. cp and BM are saved in the log c^2_3. Here, c^5_1, c^2_3, and c^3_8 have the same cp and BM.

3.4 Merge of checkpoints

[Example 3] In Figure 4, every object has a checkpoint identifier vector ⟨4, 3, 7, 2⟩. Suppose o_1 and o_4 independently initiate the checkpointing procedure. o_1 sends a checkpointed message m_1 after taking a local checkpoint c^1_5 with $\langle 5, 3, 7, 1\rangle_{1000}$, i.e. $cp = \langle 5, 3, 7, 1\rangle$ and $BM = 1000$. On receipt of m_1, o_2 takes a local checkpoint c^2_4 and then sends a checkpointed message m_2 with $\langle 5, 4, 7, 1\rangle_{1100}$. On the other hand, o_4 takes c^4_2 with $\langle 4, 3, 7, 2\rangle_{0001}$ and then sends m_4 to o_3. The object o_3 takes c^3_8 with $\langle 4, 3, 8, 2\rangle_{0011}$ and then sends m_3 to o_2. The object o_2 receives m_3 with $\langle 4, 3, 8, 2\rangle_{0011}$ from o_3 after taking c^2_4 with $cp = \langle 5, 4, 7, 1\rangle$. o_3 receives m_2 with $\langle 5, 4, 7, 1\rangle_{1100}$ after taking c^3_8 with $cp = \langle 4, 3, 8, 2\rangle$. One way is that o_2 and o_3 take c^2_5 with $\langle 4, 5, 8, 2\rangle_{0111}$ and c^3_9 with $\langle 5, 4, 9, 3\rangle_{1110}$, respectively. Here, the objects o_1, o_2, o_3, and o_4 take two checkpoints $\langle c^1_5, c^2_4, c^3_9, c^4_3\rangle$ and $\langle c^1_6, c^2_5, c^3_8, c^4_2\rangle$.

Suppose that o_4 is faulty and is rolled back to c^4_3. Then, o_3 is rolled back to c^3_9 and then o_2 is rolled back to c^2_4. Here, o_3 is required to be furthermore rolled back to c^3_8 and o_3 is also rolled back to c^4_2. In the worst case every object initiates the checkpointing procedure at the same time, 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 Figure 4, o_2 receives a checkpointed message m_3 after taking the local checkpoint c^2_4. Here, a pair of checkpoints $\langle c^1_5, c^2_4\rangle$ with $BM = 1100$ and $\langle c^3_8, c^4_2\rangle$ with $BM = 0011$ are merged into one checkpoint $\langle c^1_5, c^2_4, c^3_8, c^4_2\rangle$ with $BM = 1111$.

[Merge of checkpoints] After taking a local checkpoint c^i_t, an object o_i receives a checkpointed message m.

1. If a checkpoint c^i_u denoted by $m.cp$ is not in the same generation as c^i_t, i.e. $c^i_u.BM \cap m.BM \neq \phi$,
   - $c^i_t.cp_k := m.cp_k$ if $c^i_t.b_k = 0$ and $m.b_k =$

1 for every k ($\neq i$).

- $c_t^i.BM := c_t^i.BM \cup m.BM$.

2. Otherwise, $c_t^i.BM := c_t^i.BM \cup m.BM$ and $c_t^i.cp_k := \max(c_t^i.cp_k,\ m.cp_k)$ for every k ($\neq i$). □

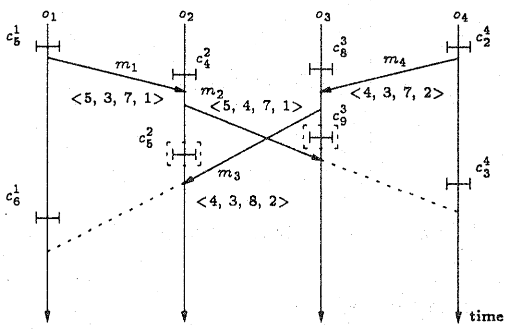

Figure 4: Checkpoints.

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

4 Restarting Protocol

If an object o_i is faulty, o_i is rolled back to the local checkpoint c_t^i which has been most recently taken. Other objects which have received influential messages sent by o_i after taking c_t^i are also required to be rolled back. In this paper, messages which o_i sends are assumed to be recorded in the sending log. The object o_i has to send a rollback request message *R-Req* to every object o_j which o_i has sent influential messages after taking c_t^i. In order to decide to which objects *R-Req* is sent, each object o_i manipulates a log SL_t^i as follows:

- When o_i takes a local checkpoint c_t^i, SL_t^i is initiated to be empty.
- Each time o_i sends an influential message m to another object o_j after c_t^i, $SL_t^i = SL_t^i \cup \{o_j\}$.

Here, m is decided to be influential according to the influential message (*IM*) condition. In order to reduce the overhead to write the log, SL_t^i is written to the log only if SL_t^i is changed. If o_i is rolled back to c_t^i, o_i sends *R-Req* to every object o_j in SL_t^i. Here, *R-Req* contains the following information:

- A checkpoint identifier vector $cp = \langle cp_1, \ldots, cp_n \rangle$ of the local checkpoint c_t^i to which o_i is rolled back.
- A rollback vector $rv = \langle rv_1, \ldots, rv_n \rangle$ where each rv_k is 1 if o_i knows o_k is rolled back to a same generation checkpoint as c_t^i, otherwise, $rv_k = 0$.

On receipt of *R-Req* from o_i, an object o_j discards *R-Req* if $R\text{-}Req.rv_j = 1$ since o_j has been already rolled back in this generation. Otherwise, $rv_k := \max(rv_k,\ R\text{-}Req.rv_k)$ ($k = 1, \ldots, n$). o_j looks for an oldest local checkpoint c_u^j where $cp_i = R\text{-}Req.cp_i$. If o_j finds such a local checkpoint c_u^j, c_u^j is referred to as *rollback point* of o_j. Otherwise, the most recent checkpoint where $cp_i < R\text{-}Req.cp_i$ becomes a *rollback point*. Then, o_j collects a set RL^j of messages which o_j has received from o_i after taking c_u^j. If there is some influential message in RL^j, o_j is rolled back to the *rollback point* c_u^j. Then, o_j sends *R-Req* to every o_k in SL_u^j with $rv_j = 1$ and $rv_k = 1$. If o_j has not received any influential message from o_i, o_j discards *R-Req* since o_j is not required to be rolled back.

5 Evaluation

We evaluate the protocol by comparing with the message-based, asynchronous protocol in terms of the number of checkpoints taken. We make the simulation on the following client-server environment:

1. There are n (≥ 1) objects $o_1, \ldots, o_n$ in the servers.
2. Transactions are initiated in a client, possibly concurrently. Each transaction issues randomly one method to the server object.
3. Each method invokes randomly methods in other objects. The maximum level of invocation is three. The level is randomly decided when a transaction invokes the method.
4. Every pair of non-update methods are compatible but every update method conflicts with every method.
5. One server object, say o_1, initiates the checkpoint procedure every time some number cn of methods are performed.

Here, let P_s denote a probability that a method invoked is a non-update type. Let $C_N(n, P_s, cn)$ and $C_O(n, P_s, cn)$ be the numbers of local checkpoints taken in the traditional way and in the O-consistent checkpoint, respectively, for n, P_s, and cn. Let $M_N(n, P_s, cn)$ and $M_O(n, P_s, cn)$ be the numbers of messages transmitted in the traditional way and the O-consistent checkpoint, respectively, for n, P_s, and cn.

In the simulation, the client initiates 800 transactions, i.e. issues 800 methods to the objects in the servers. In Figure 5, the straight line shows the ratios $C_O(n, 0.5, 2)/C_N(n, 0.5, 2)$ and the dotted line indicates $M_O(n, 0.5, 2)/M_N(n, 0.5, 2)$ for n given $P_s = 0.5$ and $cn = 2$. That is, 50% of methods invoked are non-update type. The checkpoint procedure is initiated each time every two methods are invoked in o_1. Figure 5 shows that the number of checkpoints to be taken can be reduced by taking only the object-based consistent (O-consistent) checkpoints. For example, only 60% of traditional checkpoints are taken in the O-consistent checkpoint if there are seven server objects, i.e. $n = 7$.

In Figure 6, the straight line shows $C_O(5, P_s, 2)/C_N(5, P_s, 2)$ and the dotted line shows $M_O(5, P_s, 2)/M_N(5, P_s, 2)$ for P_s, $n = 5$, and $cn = 2$. The more non-update methods are invoked, the fewer number of influential messages are transmitted and the fewer number of checkpoints are taken in the O-consistent checkpoint.

Figure 7 shows $C_O(10, 0.8, cn)/C_N(10, 0.8, cn)$ and $M_O(10, 0.8, cn)/M_N(10, 0.8, cn)$ for cn given $n = 10$ and $P_s = 0.8$. That is, 80% of the methods are non-update type. Figure 7 shows that the number of checkpoints taken by the server objects are not increased even if the checkpoint procedure is more often initiated. This means the objects which are required to be more available can often initiate the checkpointing procedure.

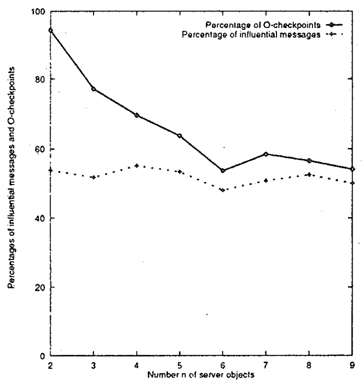

Figure 5: O-consistent checkpoints.

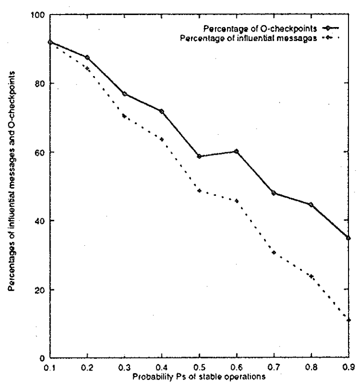

Figure 6: O-consistent checkpoints for P_s ($n = 5$).

6 Concluding Remarks

We discussed how to take *object-based consistent* (*O-consistent*) checkpoints of multiple objects, which can be taken from the application point of view but may be inconsistent with the traditional message-based definition. We defined ***influential messages*** on the basis of the conflicting relation of requests and responses 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. The *O-consistent checkpoint* is one where there is no orphan influential message. We presented the protocol for taking O-consistent checkpoints where no object is suspended in taking checkpoints. The number of local checkpoints can be reduced by the O-checkpoints.

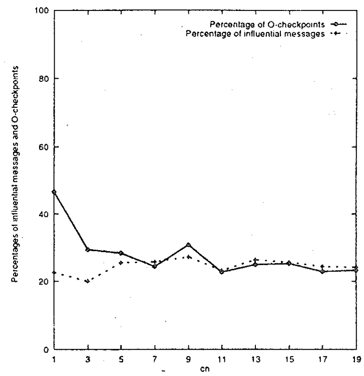

Figure 7: O-consistent checkpoints for cn ($n = 10$).

References

[1] Bhargava, B. and Lian, S. R., "Independent Checkpointing and Concurrent Rollback for Recovery in Distributed Systems — An Optimistic Approach," *Proc. of IEEE SRDS-7*, pp. 3-12, 1988.

[2] Chandy, K. M. and Lamport, L., "Distributed Snapshots : Determining Global States of Distributed Systems," *ACM TOCS*, Vol. 3, No. 1, pp. 63–75, 1985.

[3] Garcia-Molina, H., "Using Semantics Knowledge for Transaction Processing in a Distributed Database," *Proc. of ACM SIGMOD*, Vol. 8, No. 2, pp. 188-213, 1983.

[4] Koo, R. and Toueg, S., "Checkpointing and Rollback-Recovery for Distributed Systems," *IEEE TOCS*, Vol. C-13, No. 1, pp. 23–31, 1987.

[5] Leong, H. V. and Agrawal, D., "Using Message Semantics to Reduce Rollback in Optimistic Message Logging Recovery Schemes," *Proc. of IEEE ICDCS-14*, pp.227–234, 1994.

[6] Manivannan, D. and Singhal, M., "A Low-Overhead Recovery Technique Using Quasi-Synchronous Checkpointing," *Proc. of IEEE ICDCS-16*, pp.100–107, 1996.

[7] Ramanathan, P. and Shin K. G., "Checkpointing and Rollback Recovery in a Distributed System Using Common Time Base," *Proc. of IEEE SRDS-7*, pp. 13–21, 1988.

[8] Tanaka, K., Higaki, H., and Takizawa, M., "Object-Based Checkpoints in Distributed Systems," *Journal of Computer Systems Science and Engineering*, Vol. 13, No.3, pp. 125–131, 1998.

[9] Tanaka, K. and Takizawa, M., "Asynchronous Checkpointing Protocol for Distributed Object-Based Checkpoints," *Proc. of IEEE Int'l Symp. on Object-oriented Real-time Computing (ISORC'2000)*, pp. 218–225, 2000.

[10] Wang, Y. M. and Fuchs, W. K., "Optimistic Message Logging for Independent Checkpointing in Message-Passing Systems," *Proc. of IEEE SRDS-11*, pp. 147-154, 1992.