Group Protocol for Supporting Object-based Ordered Delivery

Youhei Timura, Katsuya Tanaka, and Makoto Takizawa
Tokyo Denki University
Email {timura, katsu, taki}@takilab.k.dendai.ac.jp

Distributed applications are realized by cooperation of multiple objects. A state of an object depends on in what order request and response messages are delivered in the object. In this paper, we newly define an object-based precedent relation of messages based on a conflicting relation among requests to maintain mutual consistency of the objects. Here, only the messages to be ordered in the object-based system are causally delivered. We discuss a protocol which supports the object-based ordered delivery of messages.

分散オブジェクト環境におけるメッセージの順序付けプロトコル

千村 洋平, 田中 勝也, 滝沢 誠
東京電機大学
E-mail {timura, katsu, taki}@takilab.k.dendai.ac.jp

現在の情報システムは通信網によって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分散型のシステムとなっている。本論文ではオブジェクトの意味を考慮し,必要なメッセージのみに順序を付けるグループ通信プロトコル (OBG: Object-based Group Protocol) を提案する。これにより,メッセージの処理時間と通信の負荷を削減する。

1 Introduction

In distributed systems, *groups* of multiple processes are cooperating to achieve some objectives. Many papers [2, 3, 10–14] discuss how to support a group of processes with the causally ordered (CO) / totally ordered (TO) delivery of messages at a network level. The group protocol implies $O(n^2)$ computation and communication overheads for the number n of the processes in the group. Only messages required by the applications have to be causally delivered in order to reduce the overheads.

A distributed application is realized to be a collection of cooperating objects based on the object-based framework like CORBA [16]. An object is an encapsulation of data and methods for manipulating the data where the methods are invoked by using message-passing mechanism. An application sends an object o a *request message* with a method op in order to invoke op. The method op is performed on the object o and a *response message* with the result of op is sent back. There are types of invocations, synchronous, asynchronous, and one-way ones depending on how the sender waits for the response. In addition, op may further invoke other methods, i.e. *nested invocation.*

If a pair of methods op_1 and op_2 invoked by different methods *conflict* in an object, the request messages op_1 and op_2 have to be delivered to the object in the computation order of the methods. States of the objects depend on in what order conflicting methods are performed on the objects. Thus, the *object-based ordered* (*OBO*) *relation* among request and response messages is defined based on the conflicting relation presence of the types of invocations, synchronous, asynchronous, and one-way ones. The messages received are delivered to each object in the *OBO* order, which is significant for object-based applications. A message m_1 may not precede another message m_2 in the OBO relation even if m_1 causally precedes m_2. In this paper, we present an object-based group (*OBG*) *protocol* which supports the *OBO delivery* of messages.
We can reduce the number of messages to be ordered, i.e. can reduce the communication and computation overheads in object- based systems.

2 Object-Based System

2.1 Invocation types

A *group* G is a collection of multiple objects $o_1, \ldots, o_n$ ($n > 1$) which are cooperating to achieve some objectives by exchanging messages in a network. We assume the network is asynchronous, i.e. messages sent by an object are sent to the destinations with message loss, not in the sending order, and the delay time is not bounded.

Objects are distributed in servers. A transaction in a client issues a request to an object in a server. On receipt of a request of a method op_1, op_1 is performed on an object o_1 and then the response is sent to the transaction. In fact, a thread of op is created on o_1. The thread is referred to as *instance* of op on o_1 denoted by op^1. While op_1 is being performed on the object o_1, op_1 may invoke a method op_2 on another object o_2, i.e. op_1 sends a request op_2 to o_2. op_2 is performed on o_2 and the response of op_2 is sent to op_1. Thus, the invocation is *nested.*

There are *synchronous*, *asynchronous*, and *one-way* invocations of op_2 with respect to how waits for the response of op_2 [Figure 1]. In the synchronous invocation, the method op_1 waits for a response of op_2, i.e. op_1 blocks while op_2 is being performed on o_i. This shows a remote procedure call (RPC). In the asynchronous one, op_1 is performed without blocking while eventually receiving the response of op_2. That is, op_1 and op_2 are being concurrently performed on different objects o and o_2 while op_1 eventually receives the response of op_2. In the one-way invocation, op_1 does not wait for the response of op_2 after op_2 is invoked. op_1 and op_2 are being independently performed.

There are two ways to invoke multiple methods: *serial* and *parallel* invocations. Suppose a method op invokes a pair of methods op_1 on an object o_1 and op_2 on o_2. In the serial invocation, at most one method is invoked by op at a time, e.g. op invokes op_2 after invoking op_1. If op_1 is synchronously invoked, op invokes op_2 after receiving the response of op_1. On the other hand, multiple methods can be simultaneously invoked in the parallel invocation. The requests of op_1 and op_2 are concurrently issued to the objects o_1 and o_2, respectively. Here, suppose op_1 and op_2 are synchronously invoked. The method op waits for the responses from op_1 and op_2. There are *and* and *or* ways to wait for the responses. In the *and* wait, op blocks until both of the responses are received. In the *or* wait, op starts to be performed only if at least one response is received in asynchronous and one-way invocations.

An instance can exchange data with other instances. In this paper, we assume an instance op_1^i exchanges data with op_2^j only if one of op_1^i and op_2^j invokes the other in an asynchronous or one-way manner.

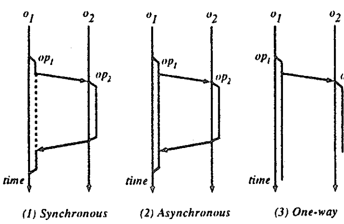


Figure 1: Types of invocation.

2.2 Conflicting methods

Let $op(s)$ denote a state obtained by performing a method op on a state s of an object o. Let $[op(s)]$ be response data of $op(s)$. A notation "$op_1 \circ op_2$" show that op_2 is serially performed after op_1 completes on an object o. "$op_1 || op_2$" shows that op_1 and op_2 are concurrently performed on o, i.e. interleaved.

A method op is referred to as *absorb* another method op_1 iff $op_1 \circ op(s) = op(s)$ and $[op_1 \circ op(s)]$for every state s of an object o. For example, a *write* method absorbs another *write* in a file object. op is referred to as *identity* iff $op(s) = s$ for every state s of o.

A pair of methods op_1 and op_2 of an object o are *compatible* iff $op_1 \circ op_2(s) = op_2 \circ op_1(s)$, $[op_1 \circ op_2(s)] = [op_2(s)]$, and $[op_1(s)] = [op_1 \circ op_2(s)]$ for every state s of o. That is, the states and the outputs obtained by performing op_1 and op_2 are independent of the computation order. op_1 and op_2 *conflict* iff they are not compatible. The *conflicting relation* among the methods is specified in the definition of the object o.

The conflicting relation is assumed to be symmetric and transitive. For example, a *counter* object supports methods *increment*, *decrement*, and *show* for increasing, decreasing, and showing the counter value, respectively. The method *increment* conflicts with *show* and *show* in turn conflicts with *increment*. The methods *increment* and *decrement* are compatible. A pair of request messages m_1 of a method op_1 and m_2 of op_2 *conflicts* iff op_1 and op_2 conflict.

Suppose a method op_t is invoked on an object o_i. An instance op_t^i is created on o_i if any method which conflicts with op_t is neither being performed nor waiting for computation on o_i. Otherwise, op_t *waits* in the wait queue of o_i.

Suppose a pair of requests op_1 and op_2 are issued to an object o_i. If op_1 and op_2 cannot be concurrently performed on o_i, op_1 and op_2 are *mutually exclusive*. The instances op_1^i and op_2^i can be *concurrent* on the object o_i if op_1 and op_2 are not mutually exclusive. op_1 and op_2 are mutually exclusive if they conflict.

Only if all the methods invoked by op complete successfully, i.e. *commit*, op commits on an object o. Otherwise, op *aborts*. Thus, each method is *atomically* performed on an object.

2.3 Precedent relation on methods

We discuss how instances are related in presence of the invocation types of methods. Suppose an instance op^i is performed on an object o_i and then completes. The response of op^i is sent back if op^i is synchronously or asynchronously invoked. Let $s(op^i)$ and $e(op^i)$ be events for invoking op on an object o_i and receiving the response of op^i, respectively. In the one-way invocation of op^i, $e(op^i)$ does not exist. The precedent relation of instances is defined on the basis of the *happen-before* relation [8]. Hence, op_1^i *precedes* op_2^i ($op_1^i \Rightarrow op_2^i$) iff $e(op_1^i)$ happens before $s(op_2^i)$ and op_1^i conflicts with op_2^i [Figure 2(1)]. op_1^i and op_2^i are concurrent ($op_1^i || op_2^i$) iff neither $op_1^i \Rightarrow op_2^i$ nor $op_2^i \Rightarrow op_1^i$.

Next, suppose a pair of methods op_1 and op_2 are performed on different objects o_j and o_k, respectively. Suppose op_1 and op_2 are invoked by an instance op^i of an object o_i. If op^i invokes op_2^k after receiving the response of op_1^j, the result of op_2^k may depend on the result of op_1^j. Hence, op_1^j *precedes* op_2^k ($op_1^j \Rightarrow op_2^k$) iff $e(op_1^j)$ happens before

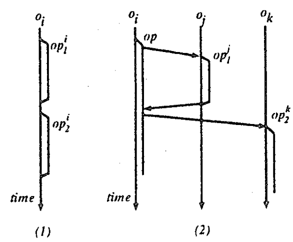


Figure 2: Invocation of op_1 and op_2

$s(op_2^k)$ [Figure 2(2)]. In the one-way and parallel invocations, $op_1^i || op_2^i$, i.e. $s(op_1^i)$ happens before

$e(op_2^k)$ and $s(op_2^k)$ happens before $e(op_1^j)$.

Now, we define the precedent relation $\Rightarrow$ of methods as follows.

[**Definition**] An instance op_1^j *precedes* another instance op_2^j ($op_1^j \Rightarrow op_2^j$) iff one of the following conditions holds:

1. op_2^j is performed before op_1^i on an object o_i ($j=i$), and op_1^i and op_2^j conflict.
2. op_1^i and op_2^j ($i \neq j$) are invoked by an instance op^k, op_1^i is synchronously or asynchronously invoked, and op_2^j is performed after op^k receives the response of op_1^j.
3. $op_1^j \Rightarrow op_3^k \Rightarrow op_2^j$ for some instance op_3^k. □

3 Object-Based Ordered Delivery

3.1 Message precedency

In the object-based system, *request*, *response*, and *data* messages are exchanged among the objects. A message m_1 *causally precedes* another message m_2 if the sending event of m_1 *happens before* the sending event of m_2 [3, 8]. A message m_1 *totally precedes* another message m_2 iff every pair of common destinations of m_1 and m_2 deliver m_1 and m_2 in the same order. Suppose an object o_i sends a message m_1 to a pair of objects o_j and o_k, and o_j sends m_2 to o_k after receiving m_1. Since m_1 causally precedes m_2, o_k has to receive m_1 before m_2. For example, if m_1 is a question and m_2 is the answer for m_1 in a teleconference, m_1 has to be delivered before m_2. Otherwise, o_k cannot understand what is discussed by o_i and o_j. However, if m_1 and m_2 are independent questions, o_k can receive m_1 and m_2 in any order. Next, suppose o_i sends a message m_1 to o_j and o_k and o_l sends m_2 to o_j and o_k. Thus, applications do not require all the messages transmitted in the network be causally and totally delivered.

We consider a *significantly precedent relation* "$\rightarrow$" among on a pair of messages m_1 and m_2 in this section, where "$m_1 \rightarrow m_2$" is meaningful for object-based applications by considering what instances send and receive m_1 and m_2. Table1 shows cases for a pair of messages m_1 and m_2 which instance op_1^i and op_2^i in an object o_i send and receive

S. An object o_i sends m_2 after m_1.

S1. m_1 and m_2 are sent by a same instance op_1^i.

S2. m_1 is sent by op_1^i and m_2 is sent by op_2^i ($op_1^i \neq op_2^i$):

S2.1. op_1^i precedes op_2^i ($op_1^i \Rightarrow op_2^i$).

S2.2. op_1^i and op_2^i are concurrent ($op_1^i || op_2^i$).

R. o_i sends m_2 after receiving m_1.

R1. m_1 and m_2 are received and sent by op_1^i.

R2. m_1 is received by op_1^i and m_2 is sent by op_2^i:

R2.1. $op_1^i \Rightarrow op_2^i$. **R2.2.** $op_1^i || op_2^i$.

T. o_i receives m_2 after m_1.

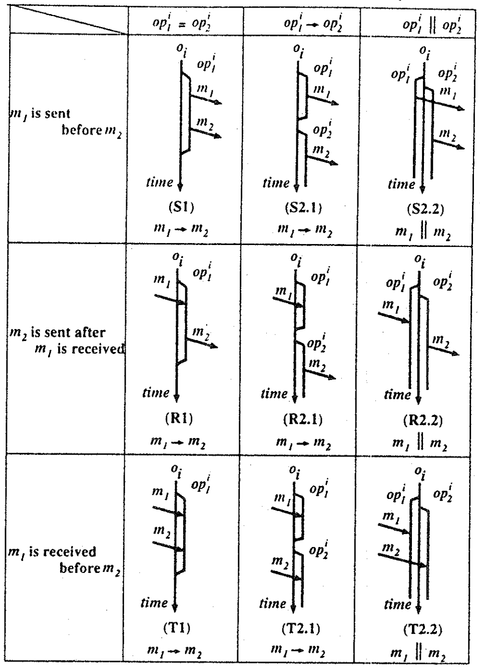
T1. m_1 and m_2 are received by an instance op_1^i.

T2. op_1^i receives m_1 and op_2^i receives m_2.

T2.1. $op_1^i \Rightarrow op_2^i$. **T2.2.** $op_1^i || op_2^i$.

The message m_1 significantly precedes m_2 ($m_1 \rightarrow m_2$) as shown in Table1. For example, in

Table 1: significant precedency



S2.1, the instances op_1^i and op_2^i are serially performed on the object o_i. If every method is synchronously or asynchronously invoked, the message m_2 is sent after op_1^i completes. Hence, it is sure that m_2 is received after m_1. If m_1 is a request to invoke a method in the one-way manner, there is possibility that m_2 is received before m_1. As shown in Table1, m_1 and m_2 have to be ordered in these cases. Next, suppose op_1^i and op_2^i conflict in T2.1. If m_1 or m_2 is a request message, m_1 has to be delivered before m_2 since $m_1 \rightarrow m_2$. Sine op_2^i is performed after op_1^i completes, op_2^i is started to be performed after receiving m_1. Hence, we do not consider a case neither m_1 nor m_2 is a request.

Following the discussions on Table1, we define the significantly precedent relation any messages as follows.

[**Definition**] A message m_1 *significantly precedes* another message m_2 ($m_1 \rightarrow m_2$) iff one of the following conditions holds:

1. An object o_i sends m_1 before m_2 and
   - **a.** a same instance sends m_1 and m_2, or
   - **b.** an instance sending m_1 conflicts with another instance sending m_2 in o_i.
2. o_i receives m_1 before sending m_2 and
   - **a.** m_1 and m_2 are received and sent by a same instance, or
   - **b.** an instance receiving m_1 conflicts with another instance sending m_2.
3. $m_1 \rightarrow m_3 \rightarrow m_2$ for some message m_3. □

[**Definition**] A message m_1 *object-based precedes* (*OB-precedes*) another message m_2 ($m_1 \preceq m_2$) iff the following condition holds :

1. m_1 significantly precedes m_2 ($m_1 \rightarrow m_2$).
2. if $m_1 || m_2$,

- m_1 and m_2 are conflicting requests and $m_1 \preceq m_2$ in every other common destination of m_1 and m_2.□

A message m_1 is referred to as *significant* for a message m_2 if $m_1 \preceq m_2$.

[OBO delivery] A distributed system supports the *object-based ordered* (OBO) delivery of messages iff every message m_1 is delivered before another message m_2 in every common destination of m_1 and m_2 if $m_1 \preceq m_2$.□

[Theorem 1] A message m_1 totally precedes another message m_2 if $m_1 \preceq m_2$.

[Proof] According to Theorem??, m_1 causally precedes m_2 if $m_1 \rightarrow m_2$. If $m_1 \parallel m_2$, m_1 and m_2 are totally preceded only if m_1 and m_2 are conflicting requests. □

In the OBO delivery, only messages to be ordered in the object-based system are delivered in the OB-precedent order $\preceq$.

4 Object-Based Group Protocol

4.1 Instance identifier

In order to consider the OB-precedent relation $\preceq$ of messages, it is critical to make clear which $op_1^i \Rightarrow op_2^j$, $op_2^j \Rightarrow op_1^i$, or $op_1^i \parallel op_2^j$ holds for every pair of instances op_1^i and op_2^j. Each instance op_t^i has two types of identifiers, *starting identifier* $sid(op_t^i)$ and *compatibility identifier* $cid(op_t^i)$, which satisfy the following properties.

- $sid(op_t^i) < sid(op_u^j)$ if a starting event $s(op_t^i)$ happens before $s(op_u^j)$.
- $cid(op_t^i) < cid(op_u^j)$ if $sid(op_t^i) < sid(op_u^j)$ and $s(op_t^j)$ happens before $e(op_t^i)$.

Here, "$cid(op_t^i) = cid(op_u^i)$" means that op_t^i and op_u^i are compatible and they are considered to be concurrently performed on an object o_i.

A variable *oid*, which is initially 0 and shows the linear clock [8], are manipulated for an object o_i as follows:

- If an instance op_t^i is initiated on o_i, $oid := oid + 1$ and $oid(op_t^i) := oid$.
- On receipt of a message from an instance op_u^j, $oid := max(oid, oid(op_u^j))$.

When op_t^i is initiated, $sid(op_t^i)$ is a concatenation of *oid* and the object number $ono(o_i)$ of o_i, $sid(op_t^i) > sid(op_u^j)$ if 1) $oid(op_t^i) > oid(op_u^j)$ or 2) $oid(op_t^i) = oid(op_u^j)$ and $ono(o_i) > ono(o_j)$.

An object vector $V = \langle v_1, \ldots, v_n \rangle$ is manipulated in an object o_i, where initially $v_j = 0$ for $j = 1, \ldots, n$, Each time an instance op_t^i is initiated in o_i, a vector V_t is created. V_t is manipulated as follows:

- If op_t^i sends a message m, $no_i := no_i + 1$ and $v_{ti} := sid(op_t^i){:}no_i$ (=$oid(op)$: $ono(o_i)$: no_i). m carries the vector $m.V$ where $m.v_j := v_{tj}$ ($j = 1, \ldots, n$). Here, v_{ti} is referred to as a global identifier of the sending operate of m in op_t^i. Let $m.id$ and $m.sid$ show a global identifier and start identifier carried by m.
- If op_t^i receives a message m from o_j, $v_{tj} := max(v_{tj}, m.v_j)$ ($j = 1, \ldots, n$).
- If op_t^i commits, $v_j := max(v_j, v_{tj})$ ($j = 1, \ldots, n$).
- If op_t^i aborts, V is not changed.

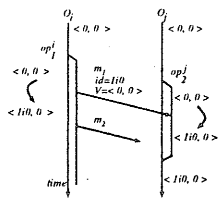


Figure 3: Object vector.

The object vectors V^i and V^j of objects o_i and o_j are initially $\langle 0, 0 \rangle$ in Figure 3. An instance op_1^i is initiated in an object o_i where a vector V_1^i is assigned to op_1^i, i.e. $V_1^i = V^i = \langle 0, 0 \rangle$. The identifier $sid(op_1^i)$ is "$1i$" which shows a concatenation $1{:}i$. op_1^i sends a request m to invoke another instance op_2^j on an object o_j. The sending event of m is identified by "$1i0$". m_1 carries the vector $V_1^i (= \langle 0, 0 \rangle)$ to the object o_j. After sending m_1, V_1^i is changed to $\langle 1i0, 0 \rangle$. On receipt of m_1, op_2^j is initiated where $sid(op_2^j) =$ "$2j$". Here, V_2^j is $\langle 1i0, 0 \rangle$. If op_2^j commits, the vector V^j of o_j is changed to V_2^j ($= \langle 1i0, 0 \rangle$). Then o_i sends a message m_2. This event is identified by "$1i1$".

[Theorem 2] A message m_1 causally precedes another message m_2 if m_1.V $<$ m_2.V.

Let us consider three objects o_i, o_j, and o_k [Figure 4]. An instance op_1^i on o_i sends a message m_1 to o_j and o_k. Instances op_2^i and op_1^i are concurrent in o_i, i.e. $op_1^i \| op_2^i$. op_2^i sends m_3 to o_k. op_3^j sends m_2 to o_k after receiving m_1. Here, m_1 significantly precedes m_2 ($m_1 \rightarrow m_2$). o_k has to receive m_1 before m_2. However, m_1 and m_3 are significantly concurrent ($m_1 \| m_3$) since $op_1^i \| op_2^i$. Similarly $m_2 \| m_3$. However, since op_3^j is initiated after receiving m_1 from op_1^i and $op_1^i \| op_2^i$, $m_1.V = m_3.V$. Hence, $m_2.V > m_3.V$. Although o_k can receive m_2 and m_3 in any order since $m_2 \| m_3$, "m_2 precedes m_3" because $m_2.V > m_3.V$. In Figure 4, since op_1^i and op_2^i are concurrent, i.e. compatible, $m_1 \parallel m_2$. However, $m_1.V < m_2.V$. In order to resolve this problem, *compatibility identifier cid* is introduced. A variable *cid*, initially 0, is manipulated as follows if an instance op^i is initiated on an object o_i :

- If no instance is being performed on o_i, $cid(op^i) := sid(op^i)$.
- Otherwise, $cid(op^i) := cid$.

If $cid(op_1^i) = cid(op_2^i)$, op_1^i and op_2^i are compatible. In Figure 4, suppose $cid = 0$ before op_1^i is initiated in o_i. $cid(op_1^i) = sid(op_1^i) =$ "$1i$" and $cid(op_2^i) =$ "$1i$" while $sid(op_1^i) < sid(op_2^i)$.

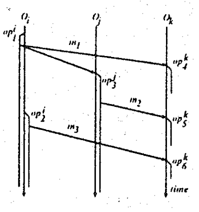


Figure 4: Message ordering.

4.2 Message ordering

In Figure 4, the instances op_3^j and op_4^k are invoked by a request m_1, op_5^k by m_2 and op_6^k by m_3. Table2 shows values of *id*, *cid*, and V of the messages. $sid(op_1^i) < sid(op_2^i)$ because op_2^i is invoked after op_1^i. Hence, $m_1.id < m_3.id$. op_1^i sends m_1 to o_j and o_k. $m_1.V = \langle 0,0,0 \rangle$ since $m_1.id =$ "$1i0$". On receipt of m_1, m_1 is enqueued into a receipt queue RQ_j of o_j. $m_1.V < m_2.V$, and $m_1.id < m_2.v_1$ and $m_1.v_2 < m_2.id$. On the other hand, $m_2.V > m_3.V$ but $m_2.id > m_2.v_2$ and $m_2.v_3 < m_3.id$.

A pair of messages m_1 and m_2 are ordered by the following rule.

[Ordering rule] A message m_1 *precedes* another message m_2 ($m_1 \Rightarrow m_2$) in a common destination of m_1 and m_2 if the following condition holds:

1. [m_1 and m_2 are sent by an object o_i]
   - the same instance sends m_1 and m_2, and $m_1.id < m_2.id$.
   - the sender instances of m_1 and m_2 conflict, i.e. $m_1.cid \neq m_2.cid$ and $m_1.id < m_2.id$,
     - m_1 or m_2 is not a request, or
     - m_1 is an asynchronous request.
2. [m_1 is sent by o_i and m_2 is sent by o_j]
   - $m_1.id \leq m_2.v_i$, $m_1.v_j \leq m_2.id$, and $m_1.v_k \leq m_2.v_k$ ($k = 1,\ldots,n$, $k \neq i$, $k \neq j$), and
     - m_1 and m_2 are conflicting requests, i.e, $m_1.op$ and $m_2.op$ conflict. □

Table 1: Object vectors.

m	$m.id$	$m.cid$	$m.V$
m_1	$1i0$	$1i$	$\langle 0,0,0 \rangle$
m_2	$2j0$	$2j$	$\langle 1i0,0,0 \rangle$
m_3	$2i0$	$1i$	$\langle 0,0,0 \rangle$

In Figure 4, op_1^i sends a request m_1 to o_j and o_k where op_3^j and op_4^k are invoked. Then, op_3^j sends a request m_2 to o_k. Here, $m_1.V < m_2.V$ and $m_1.id < m_2.v_1$ and $m_1.v_2 < m_2.id$. Suppose op_4^k conflicts with op_5^k. $m_1 \Rightarrow m_2$ since $m_1.op$ conflicts with $m_2.op$. Next, suppose op_4^k receives a data message m_2 after op_4^k is initiated by m_1. Here, $m_1 \Rightarrow m_2$ since $m_1.op = m_2.op = op_4^k$. On the other hand, $m_1.V = m_3.V$ but $m_1.id > m_3.v_1$ and $m_1.v_3 < m_3.id$. Accordingly, we check if $m_1.op$ and $m_3.op$ conflict. Since op_1^i and op_2^i are compatible, $m_1.cid = m_2.cid$.

Suppose o_k receives messages m_1 and m_3 from o_j as shown in Figure 4. Here $m_1.sid < m_3.sid$ and $m_1.cid = m_3.cid$. Hence, m_1 and m_3 are not preceded in the ordering rule.

[Theorem 3] If a message m_1 *OB-precedes* another m_2 ($m_1 \prec m_2$), $m_1 \Rightarrow m_2$. □

4.3 Message transmission and receipt

A message m includes the following fields:

$m.src$: sender object of m.
$m.dst$: set of destination objects.
$m.typ \in \{s, a, o, r, commit, abort\}$
$m.op$: method. $m.dat$: data.
$m.id$: global identifier.
$m.cid$: compatibility identifier.
$m.V = \langle V_1,\ldots,V_n \rangle$: object vector.
$m.SQ = \langle sq_1,\ldots,sq_n \rangle$: sequence numbers.

If m is a request message, $m.id$ is a global identifier of the sending event of m. $m.cid$ is a compatibility identifier of the sender instance. $m.sid$ shows the identifier of the instance which sends m and $m.no$ indicates the event number in the instance. If m is a response of a request m', $m.id = m'.id$ and $m.op = m'.op$. s, a, and o in $m.type$ indicate synchronous, asynchronous, and one-way requests, respectively. r shows response.

Variables $sq_1,\ldots,sq_n$ are manipulated for an object o_i to detect a message gap, i.e. messages lost or unexpectedly delayed. Each time o_i sends a message to another object o_j, sq_j is incremented by one. Then, o_i sends a message m to every destination in $m.dst$. o_j manipulates variables $rsq_1,\ldots,rsq_n$. rsq_i shows a sequence number of a message which o_j expects to receive next from o_i. On receipt of m from o_i, there is no gap, i.e. o_j receives every message which o_i sends to o_j before m if $m.sq_j = rsq_i$. If $m.sq_j > rsq_i$, there is a gap message m' where $m.sq_j > m'.sq_j \geq rsq_i$. That is, o_j has not yet received m' which o_i sends to o_j. o_j *correctly* receives m if o_j receives every message m' where $m'.sq_j < m.sq_j$ and $m'.src = m.src (= o_i)$. That is, o_j receives every message which o_i sends to o_j before m. If o_i does not receive a gap message m in some time units after the gap is detected, o_j requires o_i to send m again. The object o_j enqueues m in a receipt queue RQ_j even if a gap is detected on receipt of m.

When an instance op_t^i in an object o_i invokes a method op in some type t of invocation, o_i constructs a message m as follows:

$m.src := o_i$;
$m.dst :=$ set of destination objects;
$m.typ := request(t)$;
$m.op := op$;
$m.id = \langle m.sid, m.no \rangle := \langle sid(op_t^i), no_i \rangle$;
$m.cid := cid$;
$m.v_j := v_{tj}^i$ for $j = 1,\ldots,n$;
$sq_h := sq_h + 1$ for every object $o_h \in m.dst$;
$m.sq_j := sq_j$ for $j = 1,\ldots,n$;

4.4 Message delivery

The messages in a receipt queue RQ_i are ordered in the precedent order $\Rightarrow$.

[Stable message] Let m be a message which an object o_i sends to another object o_j and is stored in the receipt queue RQ_j. The message m is *stable* in o_j iff one of the following conditions holds:

1. There exists such a message m_1 in RQ_j that $m_1.sq_j = m.sq_j + 1$ and m_1 is sent by o_i.
2. o_j receives at least one message m_1 from every object, where $m \Rightarrow m_1$. □

The top message m in RQ_j can be delivered if m is stable because every message preceding m in $\Rightarrow$ is surely delivered.

[Ready message] A message m in a receipt queue RQ_j is *ready* if no method instance conflicting with $m.op$ is being performed on o_j. □

The messages in RQ_j are delivered by the following procedure.

[Delivery procedure] If the top message m in RQ_j is stable and ready, m is delivered. □

[Theorem 4] The OBG protocol delivers m_1 before m_2 if $m_1 \preceq m_2$.

[Proof] We assume that $m_1 \preceq m_2$ but m_2 is delivered before m_1. By the delivery procedure, m_2 is delivered only if m_2 is stable and ready. That is, every message $m_1 \preceq m_2$ is delivered. It contradict the assumption. □

If an object o_i sends no message to another one o_j, messages in RQ_j cannot be stable. In order to resolve this problem, o_i sends every object o_j a message without data if o_i had sent no message to o_j for some predetermined δ time units. δ is proportional to delay time between o_i and o_j. o_j considers that o_j loses a message from o_i if o_j receives no message from o_i for δ or o_j detects a message gap. o_i also considers that o_j loses a message m unless o_i receives a receipt confirmation of m from o_j in 2δ after o_i sends m to o_j. Here, o_i resends m.

5 Concluding Remarks

In this paper, we discussed how to support the object-based ordered (OBO) delivery of messages. While all messages transmitted in a network are causally or totally ordered in most group protocols, only messages to be causally ordered at the application level are ordered to reduce the delay time. Based on the conflicting relation among methods, we defined the object-based (OB) precedent relation among request and response messages. We discussed the object vector to order messages in the object-based systems.

References

[1] Ahamad, M., Raynal, M., and Thia-Kime, G., "An Adaptive Protocol for Implementing Causally Consistent Distributed Services," *Proc. of IEEE ICDCS-18*, 1998, pp.86–93.

[2] Bernstein, P. A., Hadzilacos, V., Goodman, N., "Concurrency Control and Recovery in Database Systems," *Addison-Wesley*, 1987.

[3] Birman, K., Schiper, A., and Stephenson, P., "Lightweight Causal and Atomic Group Multicast," *ACM Trans. on Computer Systems*, Vol.9, No.3, 1991, pp.272-314.

[4] Enokido, T., Tachikawa, T., and Takizawa, M., "Transaction-Based Causally Ordered Protocol for Distributed Replicated Objects," Quinton *Proc. of IEEE ICPADS'97*, 1997, pp.210–215.

[5] Enokido, T., Higaki, H., and Takizawa, M., "Group Protocol for Distributed Replicated Objects," *Proc. of ICPP'98*, 1998, pp.570–577.

[6] Enokido, T., Higaki, H., and Takizawa, M., "Protocol for Group of Objects," *Proc. of DEXA'98*, 1998, pp.470–479.

[7] Enokido, T., Higaki, H., and Takizawa, M., "Object-Based Ordered Delivery of Messages in Object-Based Systems," *Proc of ICPP'99*, 1999, pp.380–387.

[8] Lamport, L., "Time, Clocks, and the Ordering of Events in a Distributed System," *CACM*, Vol.21, No.7, 1978, pp.558–565.

[9] Mattern, F., "Virtual Time and Global States of Distributed Systems," *Parallel and Distributed Algorithms* (Cosnard, M. and , P. eds.), *North-Holland*, 1989, pp.215–226.

[10] Nakamura, A. and Takizawa, M., "Causally Ordering Broadcast Protocol," *Proc. of IEEE ICDCS-14*, 1994, pp.48–55.

[11] Ravindran, K. and Shah, K., "Causal Broadcasting and Consistency of Distributed Shared Data," *Proc. of IEEE ICDCS-14*, 1994, pp.40-47.

[12] Tachikawa, T., Higaki, H., and Takizawa, M., "Significantly Ordered Delivery of Messages in Group Communication," *Computer Communications Journal*, Vol. 20, No.9, 1997, pp. 724–731.

[13] Tachikawa, T., Higaki, H., and Takizawa, M., "Group Communication Protocol for Real-time Applications," *Proc. of IEEE ICDCS-18*, 1998, pp.40–47.

[14] Takizawa, M. and Deen, S. M., "Lock-mode Based Resolution of Uncompensatable Deadlock in Compensating Nested Transaction," *Proc. of Fareast Workshop on Future Database Systems*, 168–175, 1992.

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

[16] Object Management Group Inc., "The Common Object Request Broker : Architecture and Specification," Rev.2.1,1997.