マルチメディア通信と分散処理 88－7
（1998. 4. 24）

Protocol for A Group of Objects

Tomoya Enokido, Hiroaki Higaki, and Makoto Takizawa
Tokyo Denki University
E-mail {eno, hig, taki}@takilab.k.dendai.ac.jp

In distributed applications, a group of multiple objects cooperate. In traditional group communication protocols, messages are causally delivered at the network level. In order to reduce the protocol overhead, only messages required to be causally delivered at the application level have to be ordered. The state of the object depends on in what order the requests and responses are transmitted and the requests are computed. In this paper, we define the significant precedence order of messages based on the conflicting relation among the requests. We discuss a protocol which supports the significantly ordered delivery of request and response messages.

オブジェクトグループプロトコル

榎戸 智也　桧垣 博章　滝沢 誠
東京電機大学理工学部経営工学科

分散型のアプリケーションでは、複数のオブジェクトが互いにメッセージを交換し、協調動作を行なう。要求メッセージを受信したオブジェクトは、応答を返送する。ネットワーク層でのメッセージの因果順序配送やメッセージ紛失の無い通信を保証するグループ通信プロトコルが、これまでに議論されている。しかし、アプリケーションでは、順序付けされるべきメッセージだけが、因果順序に配送されればよい。アプリケーションを構成するオブジェクトの状態は、要求と応答の送受信順序、さらに、その要求がどのような順序で処理されるかに依存する。本論文では、演算間の競合関係をもとに、アプリケーションにとって意味のあるメッセージの先行関係を定義する。さらに、メッセージ間で意味のある先行関係を保証するプロトコルを提案する。

1 Introduction

Distributed applications like teleconferences are composed of multiple application objects. A *group* of multiple application objects have to be communicated in the distributed applications. There are two types of group communication. One type is *multicast* [2], where each object sends a message to a group of objects. The other type is *intra-group* communication [9–11] where multiple autonomous objects are cooperating. Here, the group is first defined among multiple objects. Then, messages sent by an object are delivered to the destination objects in the group. In this paper, we discuss the intra-group communication among multiple *application* objects.

Many papers [2, 5, 9–11] have discussed how to support the causally ordered delivery of messages at the network level in the presence of message loss and stop faults of the objects. $O(n^2)$ processing overhead and $O(n)$ to $O(n^2)$ communication overhead are implied for number n of objects in the group [9]. On the other hand, Cheriton *et al.* [3] point out that it is meaningless at the application level to support the causally ordered delivery of all messages transmitted in the network. Only messages required by the applications have to be causally delivered in order to reduce the overhead. Ravindran *et al.* discuss how to support the ordered delivery of messages based on the precedence relation among messages explicitly specified by the application. Agrawal *et al.* [7] define *significant* messages, on receipt of which the state of the object is changed.

An object o supports *abstract* operations for manipulating o. On receipt of a *request* message with an operation op, o computes op and sends back a response message with the result of op. The states of the objects depend on in what order the operations are computed. The *conflicting* relation [1] among the operations is defined for each object based on the semantics of the object. If two operations sending and receiving messages conflict in an object, the messages have to be received in the computation order of the operations. Thus, the *significant precedence relation* among the request and response messages can be defined based on the conflicting relation. In this paper, we present an *Object-based Group* (OG) protocol which supports the significantly preceded delivery of messages where only messages to be ordered are delivered to the application objects in the order. Tachikawa and Takizawa [13] show a protocol which uses the real time clock. However, it is not easy to synchronize real time clocks in distributed objects. In this paper, the messages are ordered by *object vectors*.

In section 2, we present the system model. In section 3, we discuss the significant precedency among messages. In section 4, the protocol for supporting the significantly ordered delivery of messages in the group is discussed.

2 System Model

The distributed application is realized by cooperation of multiple objects and by passing messages in the network. An object o can be manipulated only through an operation supported by o. Let $op(s)$ denote a state obtained by applying an operation op to a state s of o. Two operations op_1 and op_2 of an object o are *compatible* iff

$op_1(op_2(s)) = op_2(op_1(s))$ for every state s of o. op_1 and op_2 *conflict* iff they are not compatible. That is, the state obtained by applying op_1 and op_2 to o depends on the computation order of op_1 and op_2, i.e. $op_1(op_2(s)) \neq op_2(op_1(s))$ for some state s of o. The *conflicting* relation among the operations is specified when o is defined. Suppose that op_1 is issued to o while op_2 is being computed in o. If op_1 is compatible with op_2 in o, op_1 can be computed. Otherwise, op_1 has to wait until op_2 completes.

Each time o receives a request of op, a thread for op is created for o. In each thread for op, *actions* of op are computed sequentially on o. An action is a primitive unit of computation in o. The thread is an *instance* of op in o. Only if all the actions computed in op complete successfully, op completes successfully, i.e. *commits*. If some action in op fails, no action in op is computed, i.e. *aborts*. That is, op is *atomically* computed in o. op may invoke another operation op_i. op_i may further invoke operations. Thus, the computation of op is *nested*.

A *group* G is a collection of objects $o_1, \ldots, o_n$ $(n \geq 2)$ which are cooperating by sending requests and responses through the network. We assume that the network is *less reliable* and *not synchronous*, i.e. messages sent by each object are delivered to the destinations with message loss not in the sending order and the delay time among objects is not bounded.

3 Significant Precedence

3.1 Precedence of operations

It is critical to consider in what order the operations are computed in the objects. Let op_1^i and op_2^i be instances of operations op_1 and op_2 in an object o_i, respectively. Here, op_1^i *precedes* op_2^i $(op_1^i \Rightarrow_i op_2^i)$ in o_i iff op_2^i is computed after op_1^i completes. If op_1^i and op_2^i are mutually exclusive, either one of them has to precede the other. Otherwise, they can be interleaved. $op_1^i \parallel op_2^i$ shows interleaved computation of op_1^i and op_2^i.

[Definition] op_1^i *precedes* op_2^j $(op_1^i \Rightarrow op_2^j)$ iff (1) $op_1^i \Rightarrow_i op_2^j$ for $i = j$, (2) op_1^i invokes op_2^j, or (3) for some op_3^k, $op_1^i \Rightarrow op_3^k \Rightarrow op_2^j$. □

In Figure 1, $op_1^i \Rightarrow op_2^i \Rightarrow op_4^j$ and $op_1^i \Rightarrow op_3^i$, and $op_2^i \parallel op_3^i$.

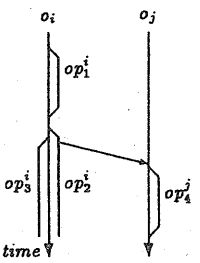

Figure 1: Precedence of operations

3.2 Significant precedence

A message m_1 *causally precedes* m_2 if the sending event of m_1 precedes the sending event of m_2 [2,6]. If m_1 is a question and m_2 is the answer of m_1, m_1 has to be received before m_2. Independent questions m_1 and m_2 can be received in any order. Thus, it is important to consider what messages are required to be ordered in the application. We define a *significant* precedence relation "$\rightarrow$" among m_1 and m_2 based on the concept of objects. There are the following cases as shown in Figures 2 and 3.

S. o_i sends m_2 after m_1.

S1. m_1 and m_2 are sent by op_1^i.

S2. m_1 and m_2 are sent by op_1^i and op_2^i, respectively:

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 interleaved.

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 and op_2^i are interleaved.

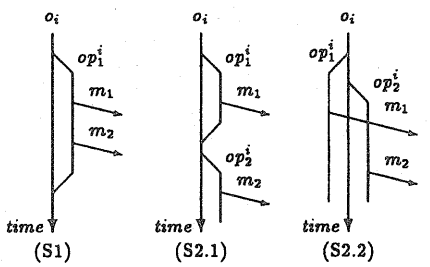

Figure 2: Send-send precedence

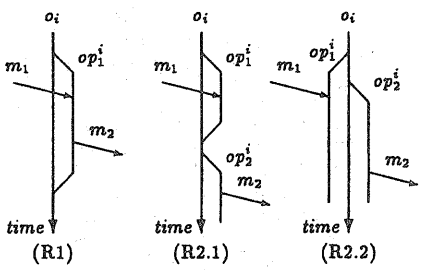

Figure 3: Receive-send precedence

In S1, m_1 *significantly precedes* m_2 $(m_1 \rightarrow m_2)$ since m_1 and m_2 are sent by the same op_1^i. In S2, m_1 and m_2 are sent by different instances op_1^i and op_2^i in o_i. In S2.1, op_1^i and op_2^i are not interleaved, i.e. op_1^i precedes op_2^i $(op_1^i \Rightarrow op_2^i)$. Unless op_1^i and op_2^i conflict, there is no relation between op_1^i and op_2^i. Hence, neither $m_1 \rightarrow m_2$ nor $m_2 \rightarrow m_1$ (written as $m_1 \parallel m_2$). Suppose op_1^i and op_2^i conflict. If $op_2^i \Rightarrow op_1^i$, the output data carried by m_1 and

m_2 may be different from $op_1^i \Rightarrow op_2^i$ because the state obtained by applying op_1^i and op_2^i depends on the computation order of op_1^i and op_2^i. Thus, if op_1^i and op_2^i conflict, the messages sent by op_1^i have to be received before the messages sent by op_2^i, i.e. $m_1 \rightarrow m_2$. Otherwise, $m_1 \parallel m_2$. In S2.2, op_1^i and op_2^i are interleaved. Since op_1^i and op_2^i are not related, $m_1 \parallel m_2$.

In R1, $m_1 \rightarrow m_2$ since m_1 is received and m_2 is sent by op_1^i. Here, m_1 is the request of op_1^i or a response of an operation invoked by op_1^i. m_2 is the response of op_1^i or a request of an operation invoked by op_1^i. For example, suppose m_1 is a response of op_2 invoked by op_1^i and m_2 is a request of op_2. The input data of op_2 may be the input of m_1.

In R2, m_1 is received by op_1^i and m_2 is sent by op_2^i ($\neq op_1^i$). In R2.1, $op_1^i \Rightarrow op_2^i$. If op_1^i and op_2^i conflict, $m_1 \rightarrow m_2$. Unless op_1^i and op_2^i conflict, $m_1 \parallel m_2$. In R2.2, $m_1 \parallel m_2$.

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

(1) m_1 is sent before m_2 by an object o_i and
  (1-1) m_1 and m_2 are sent by the same operation instance, or
  (1-2) an operation sending m_1 conflicts with an operation sending m_2 in o_i.

(2) m_1 is received before sending m_2 by o_i and
  (2-1) m_1 and m_2 are received and sent by the same operation instance, or
  (2-2) an operation receiving m_1 conflicts with an operation sending m_2.

(3) $m_1 \rightarrow m_3 \rightarrow m_2$ for some message m_3. □

[Proposition] A message m_1 causally precedes m_2 if $m_1 \rightarrow m_2$. □

According to the causality theory [8], a message m is preceded by all the messages which o_i has received and sent before o_i sends m. However, m is significantly preceded by only messages which are related with m, not necessarily all the messages.

3.3 Ordered delivery

We discuss how a message is delivered to each destination object o_i in the group G. Suppose an object o_h sends m_1 to o_i and o_j, and o_k sends m_2 to o_h, o_i, and o_j [Figure 4]. o_i and o_j receive both m_1 and m_2. There are the following cases:

C1. m_1 and m_2 are requests.

C2. One of m_1 and m_2 is a request and the other is a response.

C3. m_1 and m_2 are responses.

In C1, suppose m_1 and m_2 are requests of op_1 and op_2, respectively, where op_1 and op_2 conflict in o_i and o_j. If $m_1 \parallel m_2$, m_1 may be delivered before m_2 in o_i and m_2 before m_1 in o_j. That is, $op_1^i \Rightarrow op_2^j$ and $op_2^j \Rightarrow op_1^j$. The state of o_i obtained by the computations may be inconsistent with o_j because op_1 and op_2 conflict in o_i and o_j. In order to keep o_i and o_j consistent, m_1 and m_2 have to be delivered to o_i and o_j in the same order. For every pair of common destinations o_i and o_j of requests m_1 of op_1 and m_2 of op_2, m_1 and m_2 have to be delivered in o_i and o_j in the same order if op_1 and op_2 conflict in o_i and o_j. In C2 and C3, m_1 and m_2 are allowed to be delivered in any order.

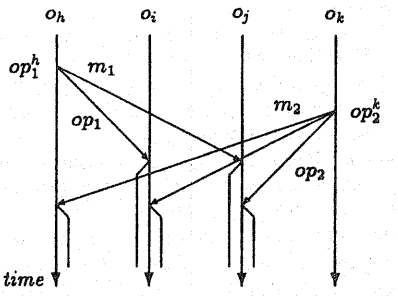

Figure 4: Receive-receive precedence

Suppose o_i receives two messages m_1 and m_2. If $m_1 \parallel m_2$ and neither m_1 nor m_2 is a request sent to multiple objects, o_i can receive m_1 and m_2 in any order. Suppose (1) $m_1 \rightarrow m_2$ and (2) m_1 or m_2 is sent to multiple objects in G. There are the following cases as shown in Figure 5.

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

T2. m_1 and m_2 are received by op_1^i and op_2^i, respectively.
  T2.1. $op_2^i \Rightarrow op_1^i$.
  T2.2. op_1^i and op_2^i are interleaved.

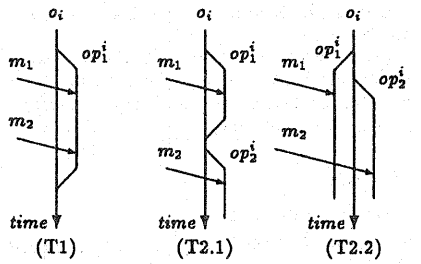

Figure 5: Receive-receive precedence

In T1, m_1 has to be delivered to o_i before m_2 since $m_1 \rightarrow m_2$. In T2, m_1 and m_2 are received by different instances. If op_1^i and op_2^i are interleaved in T2.2, m_1 and m_2 can be independently delivered to op_1^i and op_2^i. In T2.1, first suppose op_1^i and op_2^i conflict. If m_1 or m_2 is a request, m_1 has to be delivered before m_2 since $m_1 \rightarrow m_2$. Next, suppose m_1 and m_2 are responses. Unless m_1 is delivered before m_2, op_1^i waits for m_1 and op_2^i is not computed since op_1^i does not complete. That is, deadlock among op_1^i and op_2^i occurs. Furthermore, suppose m_3 is sent to op_1^i and m_4 to op_2^i and $m_4 \rightarrow m_3$. Even if op_1^i precedes op_2^i ($op_1^i \Rightarrow op_2^i$) and m_1 is delivered before m_2, the deadlock occurs because $m_4 \rightarrow m_3$. Thus, the messages destined to different instances cannot be delivered to o_i in the order $\rightarrow$ unless at least one of the messages is a request. Unless op_1^i and op_2^i conflict, m_1 and m_2 can be delivered in any order.

[Significantly ordered delivery (SO)] m_1 is delivered before m_2 in a common destination o_i of m_1 and m_2 if

(1) if $m_1 \rightarrow m_2$,
  (1-1) m_1 and m_2 are received by the same operation instance, or
  (1-2) an operation instance op_1^i receiving m_1 conflicts with op_2^i receiving m_2 in o_i, and m_1 or m_2 is a request,

(2) otherwise, if m_1 and m_2 are requests, m_1 is delivered before m_2 in the common destinations of m_1 and m_2 by the S rule. □

The condition (1-2) means that $op_1^i \Rightarrow op_2^i$ because op_1^i and op_2^i conflict and m_1 is delivered before m_2.

[Theorem] No communication deadlock occurs if every messages are delivered by the SO rule.

[Proof] Suppose m_1 is received by op_1^i and m_2 by op_2^i. If op_1^i and op_2^i are compatible or m_1 and m_2 are responses, m_1 and m_2 can be received in any order. Next, suppose op_1^i and op_2^i conflict. Suppose m_1 and m_2 are delivered by the SO rule but deadlock occurs. Since the deadlock occurs, $op_2^i \Rightarrow op_1^i$ and $m_1 \rightarrow m_2$. From (1-2) of the SO rule, $op_1^i \Rightarrow op_2^i$. It contradicts the assumption. □

[Theorem] The system is consistent if messages are delivered by the SO rule.

[Proof] Let m_1 and m_2 be messages.

(1) Suppose that $m_1 \parallel m_2$. If request messages m_1 and m_2 are sent to multiple objects and the operations of m_1 and m_2 conflict, the operations are computed in the same order by the SO rule.
(2) Suppose that $m_1 \rightarrow m_2$. By the SO rule, m_1 is delivered before m_2 if m_1 or m_2 is a request and the instances op_1^i and op_2^i receiving m_1 and m_2 conflict. Here, $op_1^i \Rightarrow op_2^i$. □

4 Protocol

4.1 Object vector

The *vector clock* [8] $V = \langle V_1, \ldots, V_n \rangle$ is used to causally order messages received. Each V_i shows a logical clock of o_i and is initially 0. o_i increments V_i by one each time o_i sends a message m. m carries the vector clock $m.V$ ($= V$). On receipt of m, o_j changes V as $V_j := \max(\ V_j,\ m.V_j\ )$ for $j = 1, \ldots, n$ and $j \neq i$. m_1 causally precedes m_2 iff $m_1.V < m_2.V$.

Significant messages are defined in context of operations instances supported by objects. That is, a group is considered to be composed of instances. The membership of the group changes if instances are initiated and terminated. If the vector clock is used, the group has to be frequently resynchronized [2, 3, 6–8, 13]. The vector clock can be used to causally order messages sent by objects but not by operation instances. In this paper, we propose an *object vector* to causally order the significant messages sent by the operation instances.

Each instance op_t^i is given a unique identifier $t(op_t^i)$ satisfying the following properties :

(1) If op_t^i starts after op_u^i in an object o_i, $t(op_t^i) > t(op_u^i)$.
(2) If o_i initiates op_t^i after receiving a request message op_t from op_u^j, $t(op_t^i) > t(op_u^j)$.

$t(op_t^i)$ is given by the linear clock [6]. o_i manipulates a variable oid showing the linear clock :

(1) Initially, $oid := 0$.
(2) $oid := oid + 1$ if an operation instance op_t^i is initiated in o_i. $oid(op_t^i) := oid$;
(3) On receipt of a message from op_u^j, $oid := \max(oid,\ oid(op_u^j))$.

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

Each action e occurring in o_i is given an event number $no(e)$ as follows :

(1) Initially, $no_i = 0$.
(2) $no_i := no_i + 1$ if e shows a changing action. $no(e) := no_i$;

Each action e in op_t^i is given a global event number $tno(e)$ as the concatenation of $t(op_t^i)$ and $no(e)$.

Each object o_i has a vector of variables $V^i = \langle\ V_1^i, \ldots, V_n^i\ \rangle$ where each V_j^i is defined for an object o_j for $j = 1, \ldots n$. Each V_j^i is initially 0. Each time an operation instance op_t^i is initiated in o_i, op_t^i is given a vector $V_t^i = \langle\ V_{t1}^i, \ldots, V_{tu}^i\ \rangle$ where $V_{tj}^i := V_j^i$. op_t^i manipulates V_t^i as follows :

(1) If op_t^i sends a message m, m carries the vector V_t^i as $m.V$ where $m.V_j := V_{tj}^i$ for $j = 1, \ldots, n$ and $V_{ti}^i := no_i$;
(2) If op_t^i receives a message m from o_j, $V_{tj}^i := m.V_j$;
(3) If op_t^i commits, $V_j^i := \max(V_j^i,\ V_{tj}^i)$ for $j = 1, \ldots, n$;

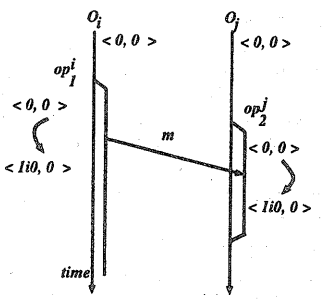

Figure 6: Object vector

Figure 6 shows two objects o_i and o_j. Initially, the vectors V_i and V_j are $\langle\ 0, 0\ \rangle$. An instance op_1^i is initiated in o_i where $V_1^i = \langle\ 0, 0\ \rangle$. After sending m to op_2^j, V_1^i is changed to $\langle\ 1i0, 0\ \rangle$ where $1i0$ is the event number of the sending of m as presented before. m carries the vector $\langle\ 1i0, 0\ \rangle$ to op_2^j. On

receipt of m, op_2^j changes V_2^j to $\langle 1i0, 0 \rangle$. After op_2^j commits, V_j of o_j is changed to be $\langle 1i0, 0 \rangle$.

4.2 Message transmission and receipt

A message m is composed of the following fields:

$m.src$ = sender object of m.
$m.dst$ = set of destination objects.
$m.type$ = message type, i.e. *request*, *responce*, *commit*, *abort*.
$m.op$ = operation.
$m.tno$ = global event number $\langle m.t, m.no \rangle$, i.e. $tno(m)$.
$m.V$ = *object vector* $\langle V_1, \ldots, V_n \rangle$.
$m.SQ$ = vector of sequence numbers $\langle sq_1, \ldots, sq_n \rangle$.
$m.d$ = data.

If m is a response message of a request m', $m.tno = m'.tno$. Here, $m.op$ denotes an operation of m'.

o_i manipulates variables $sq_1, \ldots, sq_n$. Each time o_i sends a message to o_j, sq_j is incremented by one. Then, o_i sends m to every destination o_j in $m.dst$. o_j can detect a gap between messages received from o_i by checking the sequence number sq_j, i.e. messages lost or unexpectedly delayed. o_j manipulates variables $rsq_1, \ldots, rsq_n$ to receive messages. On receipt of m from o_i, there is no gap between m and messages sent before m if $m.sq_j = rsq_i$. If $m.sq_j > rsq_i$, there is a gap m' where $m.sq_j > m'.sq_j \geq rsq_i$. That is, o_j does not receive m' which is sent by o_i. m is *correctly* received by o_j if o_j receives every message m' where $m'.sq_j < m.sq_j$. That is, o_j receives every message which o_i sends to o_j before m. o_j enqueues m in the receipt queue RQ_j.

Suppose an instance op_i^i of o_i sends a request message m of op^j. o_i constructs m as follows :

$m.src := o_i$; $m.dst :=$ set of destinations;
$m.type := request$; $m.op := op^j$;
$m.tno = \langle m.t, m.no \rangle = \langle t(op_i^i), no_i \rangle)$;
$m.V_j := V_{ij}^i \quad (j = 1, \ldots, n)$;
$sq_j := sq_j + 1$ for every o_j in $m.dst$;
$m.sq_j := sq_j \quad (j = 1, \ldots, n)$;

An additional vector $RV = \langle RV_1, \ldots, RV_n \rangle$ is given to each message m received from o_i in the receipt queue RQ_i as follows :

(1) $m.RV_i := m.tno$;
(2) $m.RV_h := m.V_h$ for $h = 1, \ldots, n$ $(h \neq i)$;

In Figure 7, op_1^i sends a message m to o_j where $m.tno = 1i0$ and $m.V = \langle 0, 0 \rangle$. On receipt of m, o_j enqueues m into RQ_j. Here, o_j gives an additional vector RV to m, i.e. $m.RV = \langle 1i0, 0 \rangle$ while $m.V$ is still $\langle 0, 0 \rangle$.

4.3 Message delivery

A pair of messages m_1 and m_2 in a receipt queue RQ_i are ordered as $m_1 \Rightarrow m_2$ according to the following ordering rule.

[Ordering rule] $m_1 \Rightarrow m_2$ if (1) or (2) holds :

(1) $m_1.V < m_2.V$ and $m_1.RV < m_2.RV$, and
  (1.1) $m_1.op = m_2.op$, or
  (1.2) $m_1.op$ conflicts with $m_2.op$.

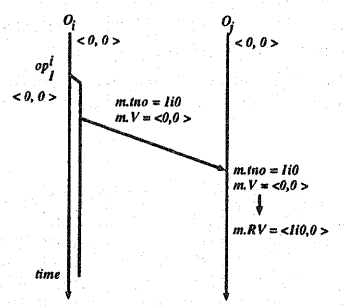

Figure 7: Receipt vector

(2) $m_1.type = m_2.type = request$ and $m_1.op$ conflicts with $m_2.op$ and $m_1.tno < m_2.tno$. □

m_1 and m_2 are *concurrent* $(m_1 \parallel m_2)$ if the ordering rule is not satisfied. Concurrent messages are stored in RQ_i in the receipt order. That is, m_1 *precedes* m_2 $(m_1 \Rightarrow m_2)$ in RQ_i if m_1 arrives at o_i before m_2. The messages received by o_i are ordered in RQ_i according to the ordering rule.

[Stable operation] Let m be a message which o_i sends to o_j and is stored in RQ_j. m is *stable* 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 such that $m \rightarrow m_1$. □

The top message m in RQ_j can be delivered if m is stable, because every message significantly preceding m is surely delivered in RQ_j.

[Definition] A message m in RQ_j is *ready* in o_j if no operation conflicting with $m.op$ is being computed in o_j. □

In addition, only significant messages in RQ_j are delivered by the following procedure in order to reduce time for delivering messages.

[Delivery procedure] While each top message in RQ_j is stable and ready, {m is delivered from RQ_j; otherwise m is neglected}. □

If an object o_i sends no message to o_j, messages in RQ_j cannot be stable. In order to resolve this problem, o_i sends o_j a message without data if o_i had sent no message to o_j for some predetermined δ time units. δ is defined to be proportional to delay time between o_i and o_j. o_j considers that o_j loses a message from o_i if (1) o_j receives no message from o_i for δ or (2) o_j detects a gap in the receipt sequence of messages. Here, o_j requires o_i to resend m. o_i also considers that o_j loses m unless o_i receives the receipt confirmation of m from o_j in 2δ after o_i sends m to o_j.

[Example] Figure 8 shows three objects o_i, o_j, and o_k. op_1^i is computed in o_i and sends a request message m_1 to o_j and o_k. On receipt of m_1, o_j computes op_2^j and o_k computes op_2^k. op_2^k sends a request message m_3 to o_i and o_j. o_i and o_j compute op_3^i and op_3^j on receipt of m_3, respectively.

op_2^j and op_3^j send response messages m_2 and m_5, respectively. op_2^k and op_3^k are interleaved in o_k. op_3^k sends a response message m_4 to o_i. Suppose op_2^j and op_3^j conflict in o_j. Each message carries $m.tno$, $m.V$, and $m.RV$ as shown in Table 1.

Table 1: Object vectors

	$m.tno$	$m.V$	$m.RV$
m_1	$1i0$	$\langle 0, 0, 0\rangle$	$\langle 1i0, 0, 0\rangle$
m_2	$2j0$	$\langle 1i0, 0, 0\rangle$	$\langle 1i0, 2j0, 0\rangle$
m_3	$2k0$	$\langle 1i0, 0, 0\rangle$	$\langle 1i0, 0, 2k0\rangle$
m_4	$3k0$	$\langle 0, 0, 0\rangle$	$\langle 0, 0, 3k0\rangle$
m_5	$3j0$	$\langle 1i0, 0, 2k0\rangle$	$\langle 1i0, 3j0, 2k0\rangle$

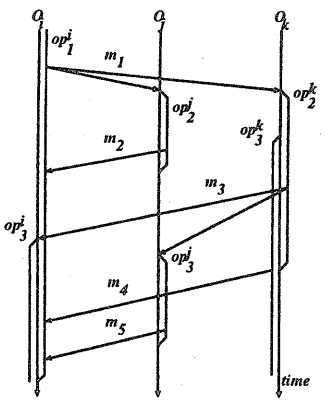

Figure 8: Example.

In the traditional group protocols, m_3 causally precedes m_4. o_i has to receive m_3 before m_4. In this paper, the messages received in a receipt queue RQ_i are ordered by using the ordering rule. Since $m_3.V(=\langle 1i0, 0, 0\rangle) > m_4.V\ (=\langle 0, 0, 0\rangle)$ but $m_3.RV\ (=\langle 1i0, 0, 2k0\rangle)$ and $m_4.RV(=\langle 0, 0, 3k0\rangle)$ are not compared, $m_3 \parallel m_4$ in o_i. Therefore, if o_i receives m_4 before m_3, o_i delivers m_4 to the application object without waiting for m_3. □

5 Concluding Remarks

In this paper, we have discussed how to support the causally ordered delivery of messages from the application point of view named the *significant precedence* while most group protocols are discussed at the network level. Only messages to be causally ordered at the application level are delivered in the significant precedence order. The system is modeled to be a collection of objects. Based on the conflicting relation among abstract operations, we have defined the significant precedence among request and response messages. We have presented a protocol for a group of objects which supports the significantly ordered delivery of messages by using the object vector.

References

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

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

[3] Cheriton, D. R. and Skeen, D., "Understanding the Limitations of Causally and Totally Ordered Communication," *Proc. of the ACM SIGOPS'93*, 1993, pp.44–57.

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

[5] Garcia-Molina, H. and Spauster, A., "Ordered and Reliable Multicast Communication," *ACM Trans. Computer Systems*, Vol.9, No.3, 1991, pp.242–271.

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

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

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

[9] Nakamura, A. and Takizawa, M., "Reliable Broadcast Protocol for Selectively Ordering PDUs," *Proc. of IEEE ICDCS-11*, 1991, pp.239–246.

[10] Nakamura, A. and Takizawa, M., "Priority-Based Total and Semi-Total Ordering Broadcast Protocols," *Proc. of IEEE ICDCS-12*, 1992, pp.178–185.

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

[12] Tachikawa, T. and Takizawa, M., "Distributed Protocol for Selective Intra-group Communication," *Proc. of IEEE ICNP-95*, 1995, pp.234–241.

[13] Tachikawa, T. and Takizawa, M., "Significantly Ordered Delivery of Messages in Group Communication," *Computer Communications Journal*, Vol. 20, 724–731, 1997.

[14] Tachikawa, T., Higaki, H., and Takizawa, M., "Group Communication Protocol for Realtime Applications," to appear in *IEEE ICDCS–18*, 1998.