Object-Based Group Communication for Distributed Systems

Takayuki Tachikawa and Makoto Takizawa
Tokyo Denki University
e-mail {tachi, taki}@takilab.k.dendai.ac.jp

In distributed applications, group communication among multiple objects is required. Group communication protocols provide a group of multiple objects with reliable and ordered delivery of messages. Kinds of group communication protocols have been discussed so far, which support the reliable and ordered delivery of messages at the communication network level. Only messages to be ordered at the application level have to be delivered in the required order. In the distributed applications, objects receive requests and send back the responses. The state of the object depends on in what order the requests are computed. In addition, the object state depend on in what order the responses are received. In this paper, we would like to discuss how to support the ordered delivery of request and response messages which are significant to the application objects.

1 Introduction

Distributed applications like teleconferences and teleclassrooms [8] are composed of multiple objects. Objects support operations for manipulating the states of the objects. Here, a *group* of multiple application objects have to be communicated in the distributed applications. First, the group is established among multiple objects. Then, messages sent by each object are delivered to the destination objects in the group. This type is *intra-group* communication [19–22, 25, 28, 29]. For example, multiple objects make a group, i.e. conference and they send messages to the objects in the group. Another type of group communication is *multicast* [3, 4], where each object sends messages to a pre-defined group or groups of objects. In this paper, we would like to discuss the intra-group communication among multiple *application* objects.

It is important to support the *causally* ordered delivery of messages in the group. Many papers [1, 3, 4, 6, 10, 11, 13, 16, 18–22, 24, 25, 28–30] have discussed how to support the reliable and ordered delivery of network messages, i.e. *packets* at the network level in the presence of message loss and object faults. $O(n^2)$ processing overhead and $O(n)$ to $O(n^2)$ communication overhead are implied for number n of objects in the group [19]. On the other hand, it is pointed out in [5] that it might be meaningless at the application level to support the causally ordered delivery of all messages transmitted in the network. Only the messages required to be causally ordered for the applications have to be causally delivered in order to reduce the communication and processing overhead. [23] discusses how to support the ordered delivery based on the precedence relation among messages specified by the application. However, it is not easy for users to specify the precedence relation. [15] defines the *significant* messages in the distributed checkpoint. That is, if the state of the object is changed on receiving a message m, m is *significant*. The receivers of the significant message m have to be rolled back if the sender of m is rolled back.

In the distributed application, each object o supports *abstract* operations for manipulating the objects. The computation on objects is based on the remote procedure call (RPC) [31]. A *request* message m with an operation op is sent to o. On receipt of m, o computes op. op might change the state of o, e.g. *write* on a *file* object. On completion of op, op sends back a response message m' with the result of op. Here, m *precedes* m' because m' is the response of m. Next, the states of the objects depend on in what order the operations are computed. The *compatibility* relation [2] among the operations is defined for each object based on the semantics of the object. That is, two operations are compatible if the same abstract state is obtained by applying them in any order. Thirdly, the responses of operations like *read* and the requests like *write* carry data from the sender object to the receiver. For example, if o sends a *write* request m after receiving a *read* response m', m' causally effects m. Thus, the *significant causal relation* among messages for the applications can be defined by the compatibility relation on the requests and the information flow relation on the requests and responses. In this paper, we would like to discuss how to support the causally ordered delivery of messages.

In section 2, we present the system model and the significant causal relation among requests and responses at the application level. In section 3 and 4, we discuss how to realize the significant causal order. In section 5, the protocol for supporting the application-oriented causally ordered delivery of messages in the group.

2 System Model

2.1 Computation model

The distributed application is composed of multiple application objects interconnected by the communication system [Figure 1]. The distributed computation of the application is composed of computations in the objects and com-

munications among the objects. An object o is defined to be a pair of abstract state D_o and a collection P_o of abstract operations for manipulating D_o. Another object o' can manipulate o only through the operations in P_o. On receipt of a *request* message of an operation *op* from o', o computes *op* and sends back the *response* of *op*. *op* may change D_o. Here, o' and o are referred to as sender and receiver of m, respectively.

A *group* G is defined to be a collection of objects $o_1, \ldots, o_n$ $(n \geq 2)$, i.e. $G = \langle o_1, \ldots, o_n \rangle$. The objects are cooperated with each other in G by sending requests and responses. For example, in the teleconferences, objects representing agents of the members make a group, i.e. conference, and they send messages to others in the group, i.e. intra-group communication [25, 26].

The communication system takes messages from the application objects and delivers them to the destination objects by using the communication networks. The communication system delivers the destinations in the causal order only the messages which have to be causally ordered from the application point of view [4] rather than all the messages. We assume that the communication network is *reliable* and *synchronous*, i.e. messages sent by each object are delivered to the destinations with no message loss in the sending order and the delay time is bounded.

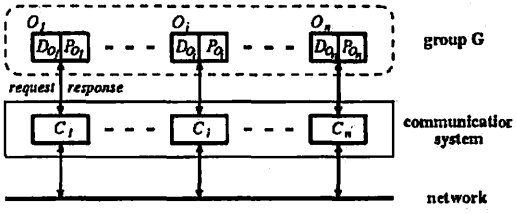

Figure 1: Group G

2.2 Conflict operations

For every operation *op* and state s of o, let $op(s)$ denote a state obtained by applying *op* to s.

[Definition] Two operations op_1 and op_2 supported by 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 are referred to as *conflict* iff they are not compatible. If op_1 and op_2 conflict, the state obtained by applying op_1 and op_2 to o depends on in which order op_1 and op_2 are computed, i.e. $op_1(op_2(s)) \neq op_2(op_1(s))$ for some state s. The *compatibility* relation $C_o \subseteq P_o^2$ for o is defined as follows: for every pair of operations op_1 and op_2 in P_o, $\langle op_1, op_2 \rangle \in C_o$ iff op_1 and op_2 are compatible. Suppose that op_2 is currently being computed in o and then op_1 is issued to o. If op_1 and op_2 are compatible in o, op_1 can be computed. If op_1 and op_2 conflict, op_1 has to wait until op_2 completes.

Multiple operations are issued to o. Some operations may be computed concurrently in o. Two operations op_1 and op_2 are referred to as *mutually exclusive* iff op_1 and op_2 cannot be computed concurrently in o. In this paper, we assume that op_1 and op_2 are mutually exclusive if op_1 and op_2 conflict.

2.3 Information flow

The request message m_1 sent by an object o_1 carries the operation op_1 and input data in_1, i.e. $m_1 = \langle op_1, in_1 \rangle$ to o_2. For example, the *Bank* object supports operations *deposit*, *withdrawal*, and *transfer*, and the *file* object supports *read* and *write*. The request of *deposit* carries the amount of money to be added to the account in the *Bank* object as the input in_1, but the request of *read* brings no input data. If $in_1 \neq \phi$, information in o_1 is flown into o_2. The response message m_2 of m_1 carries the output data of op_1, i.e. $m_2 = \langle out_2 \rangle$. For example, the response of *read* includes data derived from the file as the output out_2 but the response of *deposit* carries no output data. If the response $m_2 = \langle out_2 \rangle$ includes data, the information in o_2 is flown into o_1. If o_1 issues a request m_3 of $\langle op_3, in_3 \rangle$ after receiving m_2 which has data, m_3 may forward the information carried by m_2, i.e. m_3 is *causally effected* by m_2.

2.4 Instantiation of operation

Each time the object o receives an operation *op*, a thread for computing *op* is created if *op* is not mutually exclusive with all operations being computed or being waited in the ready queue RQ_o. If not, *op* is enqueued into RQ_o. If *op* completes, the response is sent to the sender of *op* and the thread is removed. If the top operation in RQ_o is not mutually exclusive with every operation being computed, it is started to be computed by creating the thread. Each thread for *op* is computed sequentially, i.e. a sequence of *actions*. The action is a primitive operation which is an atomic unit of computation in the object. The action cannot be directly used by the users. The computation of the thread in o is referred to as *instance* of *op* in o. The computation of *op* is viewed to be *atomic* by the sender of op_i. That is, 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* are computed, i.e. *aborts*. The completion of *op* means that *op* commits or aborts. That is, *op* can be considered to be a *transaction* [12].

op may compute an operation op_i of another object o_i. o sends the request op_i to o_i. On receipt of op_i, o_i creates the instance of op_i and op_i is computed. This action is referred to as *instantiation* of op_i. There are the following ways to instantiate op_i in *op*:

(1) Dependent instantiation:
- blocked instantiation: *op* blocks until op_i completes.
- non-blocked instantiation: *op* computed while op_i is being computed, but *op* completes after op_i completes.

(2) Independent instantiation: op_i is computed independently of *op*.

The dependent instantiation means the *remote procedure call* (RPC). op_i is instantiated in *op* and

op_i completes before op completes. On the other hand, in the independent instantiation, op_i may complete before op_i completes. In this paper, we would like to discuss the dependant instantiation.

Suppose that op in an object o_i instantiates op_i. Here, op_i may be computed in more than one object, i.e. $o_{i1}, \ldots, o_{im_i}$ ($m_i \geq 1$). Let op_i^{ij} denote an instance of op_i which is computed in o_{ij}. It is noted here that op_i^{ij} and op_i^{ik} ($j \neq k$) may be different. For example, a travel agent T issues a booking request op to a hotel object H and airline object A. op is instantiated in H as op^H to book the room and in A as op^A to book the flight. If o_{ij} and o_{ik} are replicated objects, op_i^{ij} and op_i^{ik} are the same. There are the following ways to compute op_i.

(1) Atomic computation, i.e. o_i completes only if all the instances $op_i^{i1}, \ldots, op_i^{im_i}$ complete. If some op_i^{ij} fails, op_i fails.

(2) Alternative computation, i.e. op_i completes only if at least one instance op_i^{ij} completes even if another instance op_i^{ik} fails.

(3) $\binom{n}{r}$ computation, i.e. o_i completes only if at least r ($\leq n$) instances of $op_i^{i1}, \ldots, op_i^{im_i}$ complete.

2.5 Coordination

In addition to supporting the ordered delivery of operations, it is important for applications to discuss in what order each object receives the responses. Here, suppose that an object o_i sends an operation op_i to multiple objects $o_{i1}, \ldots, o_{im_i}$. There are two approaches to sending the responses after each o_{ij} computes the instance op_i^{ij}. In one way, every o_{ij} sends back the response of op_i^{ij}, *success* or *failure*, to o_i [Figure 2(1)]. If the atomic computation of $op_i^{i1}, \ldots, op_i^{im_i}$ is required, o_i sends the *commit* message to every o_{ij} if o_i receives the response *success* of op_i^{ij} from every o_{ij}. If o_i receives *failure* from some o_{ik}, o_i sends *abort*. This is the famous *two-phase commitment* protocol [12]. The sender o_i plays a role of the centralized controller. Since only o_i receives the responses from all the receivers, o_i can decide on the receipt order of the responses.

In another way, each o_{ij} sends the response to o_i and the other receivers $o_{i1}, \ldots, o_{im_i}$ if op_i^{ij} completes [Figure 2(2)]. Here, o_{ij} knows how the other objects compute the instances of op_i. It is the distributed control. The sender o_i and each receiver o_{ij} may receive the responses not in the same order. For example, suppose that three objects A, B, and C which are agents of persons are holding the conference. Each object has the schedule of the person. One object, say A would like to hold a meeting with B and C. A sends the request of the meeting to B and C. B and C change the schedules on receipt of the request, and sends the responses to all the objects. Each object can decide on the meeting by itself if it receives all the responses. In addition, each object may compute operations based on the receipt order of the responses. Here, each object has to receive the responses in the same order.

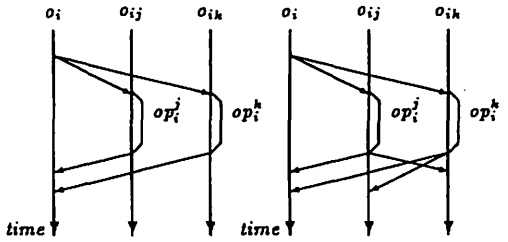

Figure 2: Atomic computation

3 Significant Precedence of Operations

It is important to consider in what order the operations are computed in the objects. The following notations for an operation op are used for an object o_i in this paper.

- op^i = instance of op in o_i.
- $[op^i$ = begin action in op^i.
- $op^i]$ = end action in op^i.
- $(op^i$ = instantiation action of op, i.e. op is started to be computed in o^i.
- $op)^i$ = completion of instantiation of op in o_i.
- $(op)^i = (op^i$ and $op)^i$, i.e. op is instantiated and completed in o_i.

The instance op^i is modeled as sequences of actions in o_i. Each action is computed atomically in o_i. The *local precedence* relation "$\rightarrow_i$" among the actions in o_i is defined as follows.

[Definition] For every pair of actions a_1 and a_2 supported by object o_i, a_1 *precedes* a_2 in o_i ($a_1 \rightarrow_i a_2$) iff a_2 is computed after a_1 o_i. □

"$\rightarrow_i$" is transitive. a_1 and a_2 are computed *concurrently* iff neither $a_1 \rightarrow_i a_2$ nor $a_2 \rightarrow_i a_1$. It is trivial that $[op^i \rightarrow_i op^i]$. If op_2 is instantiated in op_1^i, $[op_1^i \rightarrow_i (op_2^i \rightarrow_i op^i]$. If op_2 is dependently instantiated in op_1^i, $[op_1^i \rightarrow_i (op_2)^i \rightarrow_i op^i]$, i.e. op_2 is computed by the remote procedure call in op_1.

Now, we would like to discuss the precedence relation among the operation instances in o_i. Here, let op_1^i and op_2^i be instances of operations op_1 and op_2 computed in o_i, respectively. There are the following transitive precedence relations from op_1^i to op_2^i [Figure 3].

(1) op_1^i *fully precedes* op_2^i ($op_1^i \Rightarrow_i op_2^i$) iff $op_1^i] \rightarrow_i [op_2^i$.

(2) op_1^i *top-precedes* op_2^i in o_i ($op_1^i \mapsto_i op_2^i$) iff $[op_1^i \rightarrow_i [op_2^i$.

(3) op_1^i *tail-precedes* op_2^i in o_i ($op_1^i \hookrightarrow_i op_2^i$) iff $op_1^i] \rightarrow_i op_2^i]$.

(4) op_1^i *dependently instantiates* op_2^i in o_i ($op_1^i \models_i op_2^i$) iff $[op_1^i \rightarrow_i (op_2^i) \rightarrow_i op_1^i]$.

(5) op_1^i *independently instantiates* op_2^i in o_i ($op_1^i \vdash_i op_2^i$) iff $[op_1^i \rightarrow_i (op_2^i \rightarrow_i op_1^i]$.

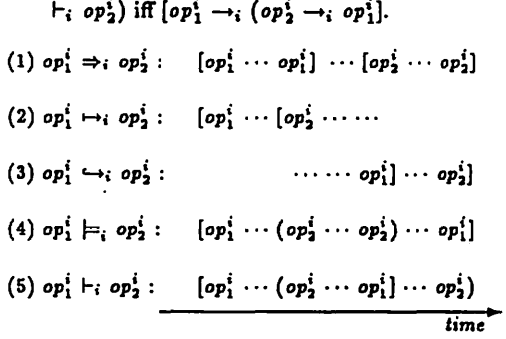

Figure 3: Precedence relation among operations

$op_1^i \Rightarrow_i op_2^i$ means that op_2^i is started after op_1^i completes in o_i, If op_1^i and op_2^i are mutually exclusive, one of them has to fully precede the other. Otherwise, they can be computed concurrently. $op_1^i \mapsto_i op_2^i$ means that op_1^i is started before op_2^i but does not necessarily mean that $op_1^i \Rightarrow_i op_2^i$. $op_1^i \hookrightarrow_i op_2^i$ means that op_1^i completes before op_2^i. Hence, $op_1^i \mapsto_i op_2^i$ and $op_1^i \hookrightarrow_i op_2^i$ if $op_1^i \Rightarrow_i op_2^i$. op_1^i *partially precedes* op_2^i if $op_1^i \mapsto_i op_2^i$ or $op_1^i \hookrightarrow_i op_2^i$. If $[op_1^i \rightarrow_i [op_2^i \rightarrow_i op_1^i]$, op_1^i and op_2^i are *interleaved* in o_i. If op_1^i top-precedes op_2^i and op_2^i tail-precedes op_1^i, i.e. $[op_1^i \rightarrow_i [op_2^i \rightarrow_i op_2^i] \rightarrow_i op_1^i]$, op_2^i is *included* in op_1^i. $op_1^i \models_i op_2^i$ and $op_1^i \vdash_i op_2^i$ mean that op_2^i is instantiated in op_1^i.

Next, we would like to consider the full precedence relation "$\Rightarrow$" among the operations computed in different objects.

- $op_1^i \Rightarrow op_2^j$ iff (1) $op_1^i \Rightarrow_i op_2^j$ for $i = j$, (2) $op_1^i] \rightarrow_i (op_2)^i$, or (3) for some op_3^k, $op_1^i \Rightarrow op_3^k \Rightarrow op_2^j$.

$op_1^i \Rightarrow op_2^j$ means that op_1^i completes before op_2^j starts. In Figure 4, o_i computes op_1^i, instantiates op_2 computed in o_j, and computes op_3^i, i.e. $op_1^i] \rightarrow_i op_2^i \rightarrow_i [op_3^i$. Here, $op_1^i \Rightarrow op_3^i$ and $op_1^i \Rightarrow op_2^j$. In this paper, we would like to discuss the full precedence relation among the operations.

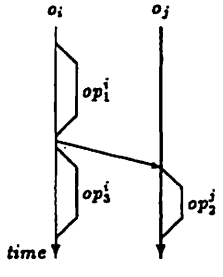

Figure 4: Full precedency

As presented before, if two operations op_1 and op_2 conflict in an object o, the state obtained by applying op_1 and op_2 to o depends on the computation order of op_1 and op_2. The significant precedence relation $\prec$ between op_1^i and op_2^i is defined as follows [Figure 5].

[Definition] op_1^i *significantly precedes* op_2^j ($op_1^i \prec op_2^j$) iff $op_1^i \Rightarrow op_2^j$, and

(1) op_1^i and op_2^j conflict in o_i and $op_1^i \Rightarrow_i op_2^j$ for $i = j$,

(2) there is some op_3^i such that $op_1^i \Rightarrow_i op_3^i$ and $op_3^i \vdash_i op_2$ (op_3^i instantiates op_2), or

(3) there is some op_3^k such that $op_1^i \prec op_3^k \prec op_2^j$. □

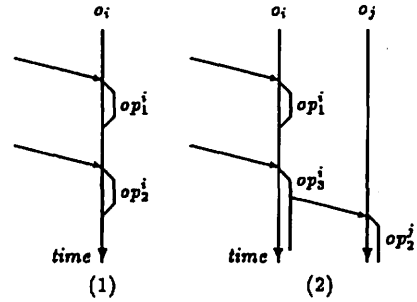

Figure 5: Significant precedency ($op_1^i \prec op_2^j$)

4 Significant Precedence of Messages

The communication system delivers messages to the destinations in the group so that every object in the group can compute actions specified by the messages in the receipt order.

4.1 Message types

There are two kinds of messages, i.e. *request* and *response* ones as presented before. The request message includes the operation *op* and the input *in* of *op*. The response message includes the output *out* of *op*. If a message m carries data as the input or output, information in the sender of m is flown into the receiver. Hence, it is important to consider whether the messages carry data or not. For example, the request of *deposit* carries the amount of money as the input.

The second point is concerned with whether the operation *op* of the request message changes the state of the object or not. For example, *read* does not change the state of the *file* object while *deposit* changes the state of the *Bank* object.

Thus, the request message $m = \langle op, in \rangle$ is typed as $\alpha\beta$-request where α is S if *op* changes the state of the object, N otherwise, and β is I if $in \neq \phi$, N otherwise. For example, the request of *read* is NN, and *deposit* is SI. The response message $m = \langle out \rangle$ is typed as γ-response where γ is O if $out \neq \phi$, N otherwise. For example, the response of *read* is O, and *deposit* is N.

4.2 Message precedence in object

We would like to discuss the precedence relation "$\prec_i$" among messages sent and received by an object o_i. There are three cases, (1) o_i sends m_1 before m_2, (2) o_i sends m_2 after receiving m, and (3) o_i receives m_1 before m_2.

4.2.1 Send-send precedence

First, suppose that o_i sends m_1 before m_2. There are the following cases:

S1. m_1 and m_2 are sent by the same instance op_1^i [Figure 6(1)].

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

S2.1. $op_1^i \Rightarrow op_2^i$ [Figure 6(2.1)].

S2.2. op_1^i and op_2^i are interleaved [Figure 6 (2.2)].

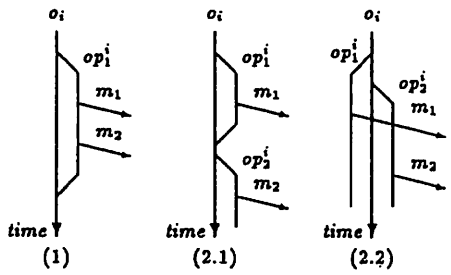

Figure 6: Send-send precedence

In S1, m_1 *precedes* m_2 in o_i ($m_1 \prec_i m_2$).

In S2, if m_1 and m_2 carry no data, m_1 and m_2 are not ordered ($m_1 \parallel_i m_2$). Here, $m_1 \parallel_i m_2$ means that neither $m_1 \prec_i m_2$ nor $m_2 \prec_i m_1$. If m_1 or m_2 carries data, the information is flown out from o_i. In S2.1, if op_1^i and op_2^i conflict in o_i and m_1 or m_2 carries data, $m_1 \prec_i m_2$ if $op_1^i \Rightarrow op_2^i$. Because the data carried by m_1 or m_2 depend on the computation order of op_1^i and op_2^i. Table 1 shows the precedence relation in case S2.1 where ○ means "$m_1 \prec_i m_2$" if op_1^i and op_2^i conflict and - means "$m_1 \parallel_i m_2$". If m_1 and m_2 are responses of op_1^i and op_2^i, respectively, $m_1 \prec_i m_2$ if m_1 and m_2 carries data.

In S2.2, op_1^i and op_2^i are interleaved. Here, $m_1 \parallel_i m_2$.

4.2.2 Receive-send precedence

Next, suppose that o_i sends m_2 after receiving m_1. There are the following cases [Figure 7]:

R1. m_1 and m_2 are communicated in the same instance op_1^i.

R2. m_1 is received in op_1^i and m_2 is sent in op_2^i ($\neq op_1^i$):

R2.1. $op_1^i \Rightarrow op_2^i$.

R2.2. op_1^i and op_2^i are interleaved.

$m_1 \backslash m_2$	IS	IN	NS	NN	O	N
IS	○	-	○	-	○	-
IN	-	-	-	-	-	-
NS	○	-	○	-	○	-
NN	-	-	-	-	-	-
O	-	-	-	-	○	-
N	-	-	-	-	-	-

Table 1: Send - send (2.1)

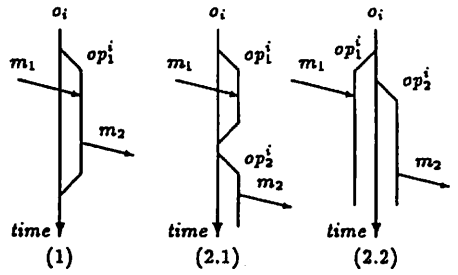

Figure 7: Receive-send precedence

If m_1 is the response without data, i.e. *N*-response, there is no precedence relation between m_1 and m_2.

Here, suppose that m_1 is the request of op_1^i or *O*-response. In case R1, if m_2 does not include data, m_1 does not precede m_2 ($m_1 \parallel_i m_2$). Hence, m_1 precedes m_2 in o_i ($m_1 \prec_i m_2$) if m_1 is *O*-response, or m_1 is the request of op_1^i, and m_2 includes data or the response of op_1^i. Table 2 shows "$\prec_i$" between m_1 and m_2 in o_i for R1. △ shows that $m_1 \prec_i m_2$ if m_1 is the request of op_1^i and m_2 is the response of op_1^i.

In R2.1, if op_1^i and op_2^i conflict in o_i and m_2 carries data, the data carried by m_2 may be derived from the data changed by op_1^i. Here, if m_1 is the request of op_1^i or *O*-response, $m_1 \prec_i m_2$ if op_1^i and op_2^i conflict and m_2 carries data. Table 3 shows "$\prec_i$" between m_1 and m_2 for R2.1 where op_1^i and op_2^i conflict.

In R2.2, op_1^i and op_2^i are interleaved. Here, $m_1 \parallel_i m_2$.

$m_1 \backslash m_2$	IS	IN	NS	NN	O	N
IS	○	○	○	-	○	△
IN	○	○	-	-	○	△
NS	○	-	○	-	△	△
NN	-	-	-	-	△	△
O	○	○	-	-	○	-
N	-	-	-	-	-	-

Table 2: Receive - send (1)

$m_1 \backslash m_2$	IS	IN	NS	NN	O	N
IS	○	○	○	-	○	-
IN	-	-	-	-	-	-
NS	○	-	○	-	○	-
NN	-	-	-	-	-	-
O	-	-	-	-	-	-
N	-	-	-	-	-	-

Table 3: Receive - send (2.1)

4.2.3 Receive-receive precedence

Let us consider a case that messages are sent to multiple objects. Suppose that an object o_j sends a message m_2 after receiving m_1 and o_k receives m_1 and m_2 as shown in Figure 8. Here, suppose that $m_1 \prec_j m_2$. Problem is in which order o_k has to receive m_1 and m_2. There are the following cases on types of messages m_1 and m_2:

C1. m_1 and m_2 are requests.
C2. m_1 is a request and m_2 is a response.
C3. m_1 is a response and m_2 is a request.
C4. m_1 and m_2 are responses.

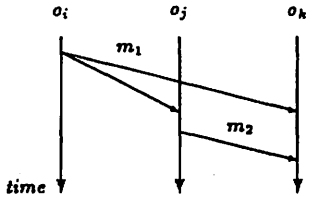

Figure 8: Receive-receive precedence (1)

In addition, there are two points on how m_1 and m_2 are received in o_k.

(a) m_1 and m_2 are received by a same instance op_1^k.
(b) m_1 and m_2 are received by different instances op_1^k and op_2^k, respectively.

For case (a), only C2 and C4 can be considered because m_2 must not be a request. Here, m_1 has to precede m_2, i.e. $m_1 \prec_k m_2$.

Next, let us consider case (b) as shown in Figure 9. In C1, suppose that m_1 and m_2 are the request of op_1^k and op_2^k. If op_1^k and op_2^k conflict in o_k, op_1^k has to fully precede op_2^k, i.e. $op_1^k \Rightarrow op_2^k$. Here, $m_1 \prec_k m_2$. Otherwise, $m_1 \parallel_k m_2$.

In C2, m_1 is a request of op_1^k. If op_1^k and op_2^k conflict in o_k, op_1^k has to fully precede op_2^k, i.e. $op_1^k \Rightarrow op_2^k$. Hence, m_2 is received after m_1, i.e. $m_1 \prec_k m_2$. If not conflict, $m_1 \parallel_k m_2$.

In C3, m_2 is a request of op_2^k. Like C2, $m_1 \prec_k m_2$ if op_1^k and op_2^k conflict in o_k. Otherwise, $m_1 \parallel_k m_2$.

In C4, the responses m_1 and m_2 are received by op_1^k and op_2^k, respectively. Unless op_1^k and op_2^k conflict in o_k, $m_1 \parallel_k m_2$. Suppose that op_1^k and op_2^k conflict in o_k. If m_1 and m_2 are N-responses, i.e. without data, $m_1 \parallel_k m_2$. If not, $m_1 \prec_k m_2$. This requires that $op_1^k \Rightarrow op_2^k$. However, if op_2^k starts before op_1^k and waits for m_2, op_2^k has to wait indefinitely because m_2 is delivered to o_k after m_1. That is, the communication deadlock occurs. In this case, op_2^k has to be aborted by the time out or deadlock resolution mechanism.

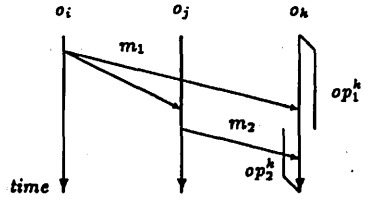

Figure 9: Receive-receive precedence (2)

4.3 Message precedence among objects

Suppose that messages m_1 and m_2 are sent to multiple objects and there are multiple common destination objects of m_1 and m_2. Suppose that o_h sends m_1 to o_i, o_j, and o_k, and o_k sends m_2 to o_h, o_i, and o_j. o_i and o_j receive both m_1 and m_2. Problem is in which order o_i and o_j receive m_1 and m_2. There are four cases on types of messages m_1 and m_2 as discussed in the receive-receive precedence. In C1, suppose that m_1 and m_2 are requests of op_1 and op_2, respectively [Figure 10]. The common destination objects o_i and o_j receive m_1 and m_2. If op_1^i and op_2^i conflict in o_i, and op_1^j and op_2^j conflict in o_j, then $op_1^i \Rightarrow op_2^i$ iff $op_1^j \Rightarrow op_2^j$ to realize the serializability [2].

In C4, m_1 and m_2 are responses. Suppose that m_1 and m_2 are the responses of operations op_1 and op_2, respectively. The common destination objects o_i and o_j receive m_1 and m_2 in the same order.

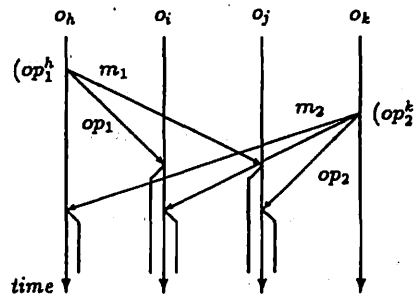

Figure 10: Receive-receive precedence (3)

5 Protocol

We would like to present a protocol for supporting the significantly causally ordered delivery

of messages.

5.1 Assumptions

The communication system delivers the messages to the application objects so that each application object o_i can receive the request and response messages in the application-oriented causal precedence order $\prec_i$. In this paper, we assume that the communication network is reliable and synchronous, i.e. the loss-less, FIFO delivery is supported and the communication delay is bounded to be δ. According to the advances of the hardware technologies, each object can have the real-time clock. Hence, we assume that each object o_i has the real-time clock C_i which shows the same global time. o_i has a variable T_i which denotes the current value of C_i. Each time o_i initiates the computation of op^i, the value of T_i is given to op^i as its *time stamp* $ts(op^i)$.

Each time o_i sends a message m, o_i gives the value of T_i to m as the time stamp $ts(m)$. Hence, for every pair of messages m_1 and m_2, we assume either $ts(m_1) < ts(m_2)$ or $ts(m_1) > ts(m_2)$.

According to the assumptions, every destination object o_i of each message m sent by o_j receives m in δ time units after o_j sends m, i.e. o_i receives m at time t where $ts(m) < t \leq ts(m) + \delta$. It is sure that every message m' causally preceding m ($m' \prec_j m$) is sent by o_j before $ts(m)$. Hence, the following theorem holds.

[Theorem] Every object o_i can receive every message m' causally preceding m, i.e. $m' \prec_i m$, until $ts(m) + \delta$. □

Following this theorem, o_i can deliver every message m sent by o_j in the order $\prec_i$ at $ts(m) + \delta$. However, m has to stay in the queue of o_i for δ. In this paper, we present a scheme in which each object o_i can deliver m to the application object in δ time units after o_j sends m.

5.2 Transmission and receipt

We would like to present the data transmission procedure by which each object o_i in the group $G = \langle o_1, ..., o_n \rangle$ can deliver the messages in $\prec_i$. For each message m, let $type(m)$ denote the type of m, i.e. $type(m) \in \{ IS, NS, IN, NN, O, N \}$, and $send(m)$ and $rec(m)$ be the sender and receiver of m.

First, suppose that o_j sends a message m to o_i. On receipt of m, o_i enqueues m into a receipt queue RQ_i. Since the network supports the FIFO delivery, it is sure that $ts(m_1) < ts(m_2)$ if m_1 and m_2 are received from the same object and m_1 precedes m_2 in RQ_i. As presented in the preceding subsection, all the messages in RQ_i whose time stamps are smaller than or equal to $t - \delta$ can be delivered in the time stamp order at time t. However, the message m has to stay in the queue until $ts(m) + \delta$. We would like to present a method by which every message m can be delivered for shorter time than δ.

Each message m carries a vector of time values $P\_clock = \langle t_1, ..., t_n \rangle$. Let $P\_clock_k(m)$ denote the k-th element t_k of P_clock in m ($k = 1, ..., n$). Here, suppose that m is sent by o_i, i.e. $send(m) = o_i$. Each object o_i has a clock stack CS_{ij} which is a stack of message type and vector clock piggybacked by messages sent by o_j. First, o_i stores the value of T_i in $P\_clock_i(m)$ which denotes $ts(m)$. For each o_j ($\neq o_i$), o_i finds messages which precede m in $\prec_i$ by comparing the message types in CQ_{ij} with $type(m)$ by using Table 2. Then, a message m_j where $ts(m)$ is the maximum among these messages is selected. $ts(m_j)$ is stored in $P\_clock_j(m)$. Then, o_i sends m. If the receiver o_j receives m from o_i, o_j knows what messages precede m in $\prec_i$.

Next, let us consider how o_i delivers messages received in RQ_i. Suppose that o_i receives a message m from o_j. Here, m is enqueued into RQ_i and then RQ_i is sorted in the time stamp order. If m in RQ_i satisfies the following delivery rule, m is delivered to the application.

[Delivery rule] For every o_j ($\neq o_i$),

(1) every message m' where $P\_clock_j(m) \geq P\_clock_j(m')$ has already been delivered, or

(2) $P\_clock_j(m) < T_i - \delta$. □

Suupose that m is delivered by the delivery rule. The information on m, i.e. $\langle type(m), P\_clock_j(m) \rangle$ is enqueued into CQ_{ij} for $j = 1, ..., n$. At each time t, the tuple $\langle type, time \rangle$ in $CQ_{i1}, ..., CQ_{in}$ is removed if $time < t - \delta$. Because messages whose *time* stamp is smaller than $t - \delta$ have already been delivered in all the destination objects.

5.3 Operations

Suppose that an operation op^i is started in o_i. First, $CQ_{i1}, ..., CQ_{in}$ are copied to $CQ'_{i1}, ..., CQ'_{in}$. op^i uses $CQ'_{i1}, ..., CQ'_{in}$ to send and receive messages. On completion of op^i, only tuples whose types are IS or NS (from Table 3) in $CQ'_{i1}, ..., CQ'_{in}$ and which are not in $CQ_{i1}, ..., CQ_{in}$ are stored in $CQ_{i1}, ..., CQ_{in}$.

6 Concluding Remarks

In this paper, we have discussed how to support the causally ordered delivery of messages from the application point of view while most group communication protocols discuss it at the network level. Only messages to be causally ordered at the application level are causally ordered. The system is modeled to be a collection of objects in this paper. Based on the compatibility relation among the operations supported by each object, the meaningful order is decided.

References

[1] Amir, Y., Dolev, D., Kramer, S., and Malki, D., "Transis: A Communication Sub-System for High Availability," *Proc. of IEEE 22th Annual Int'l Symp. on Fault-Tolerant Computing*, 1993, pp.76–84.

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

[3] Birman, K. P. and Joseph, T. A., "Reliable Communication in the Presence of Failures,"

ACM Trans. on Computer Systems, Vol.5, No.1, 1987, pp.47-76.

[4] 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.

[5] 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.

[6] Chang, J. M. and Maxemchuk, N. F., "Reliable Broadcast Protocols," *ACM Trans. Computer Systems*, Vol.2, No.3, 1984, pp.251–273.

[7] Denning, D. E., "Cryptography and Data Security," *Addison-Wesley*, 1982.

[8] Ellis, C. A., Gibbs, S. J., and Rein, G. L., "Groupware," *Comm. ACM*, Vol.34, No.1, 1991, pp.38–58.

[9] Eswaren, K. P., Gray, J., Lorie, R. A., and Traiger, I. L., "The Notion of Consistency and Predicate Locks in Database Systems," *CACM*, Vol.19, No.11, 1976, pp.624–637.

[10] Garcia-Molina, H. and Spauster, A., "Message Ordering in a Multicast Environment," *Proc. of IEEE ICDCS-9*, 1989, pp.354–361.

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

[12] Gray, J., "Notes on Database Operating Systems," *Operating Systems: An Advanced Course*, (Bayer, R. ed.), *Springer-Verlag*, 1978.

[13] Kaashoek, M. F., Tanenbaum, A. S., Hummel, S. F., and Bal, H. E., "An Efficient Reliable Broadcast Protocol," *ACM Operating Systems Review*, Vol.23, No.4, 1989, pp.5–19.

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

[15] 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.

[16] Luan, S. W. and Gligor, V. D., "A Fault-Tolerant Protocol for Atomic Broadcast," *IEEE Trans. Parallel and Distributed Systems*, Vol.1, No.3, 1990, pp.271–285.

[17] 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.

[18] Melliar-Smith, P. M., Moser, L. E., and Agrawala, V., "Broadcast Protocols for Distributed Systems," *IEEE Trans. on Parallel and Distributed Systems*, Vol.1, No.1, 1990, pp.17-25.

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

[20] Nakamura, A. and Takizawa, M., "Design of Reliable Broadcast Communication Protocol for Selectively Partially Ordered PDUs," *Proc. of IEEE COMPSAC'91*, 1991, pp.673–679.

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

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

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

[24] Schneider, F. B., Gries, D., and Schlichting, R. D., "Fault-Tolerant Broadcasts," *Science of Computer Programming*, Vol.4, No.1, 1984, pp.1–15.

[25] Tachikawa, T. and Takizawa, M., "Selective Total Ordering Broadcast Protocol," *Proc. of the 2nd IEEE ICNP*, 1994, pp.212–219.

[26] Tachikawa, T. and M. Takizawa, "Multimedia Intra-Group Communication Protocol," *Proc. of 4th IEEE International Symp. on High Performance Distributed Computing (HPDC-4)*, 1995, pp.180–187.

[27] Tachikawa, T., and Takizawa, M., "Distributed Protocol for Selective Intra-group Communication," to be appear in *Proc. of the 3rd IEEE International Conf. on Network Protocols (ICNP-95)*, *Tokyo* 1995,

[28] Takizawa, M., "Cluster Control Protocol for Highly Reliable Broadcast Communication," *Proc. of IFIP Conf. on Distributed Processing*, 1987, pp.431–445.

[29] Takizawa, M. and Nakamura, A., "Partially Ordering Broadcast (PO) Protocol," *Proc. of IEEE INFOCOM'90*, 1990, pp.357–364.

[30] Verissimo, P., Rodrigues, L., and Baptista, M., "AMp: A Highly Parallel Atomic Multicast Protocol," *Proc. of the ACM SIGCOMM'89*, 1989, pp.83-93.

[31] Wood, M. D., "Replicated RPC Using Amoeba Closed Group Communication," *Proc. of IEEE ICDCS-13*, 1993, pp.499-507.