Object-Based Causality in Group Communication *

2E−4

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

1 Introduction

Distributed applications like teleconferences are composed of multiple objects. Objects support operations for manipulating the states of the objects. A *group* is a collection of multiple application *objects*. After the group is established, messages sent by each object are delivered to the destinations in the group.

We have to find the significant precedence relation among messages from the application point of view. The computation on objects is based on the remote procedure call (RPC). The *compatibility* relation among the operations is defined for each object based on the semantics of the object. The responses of operations like *read* and the requests of operations like *write* carry data from the sender object to the receivers. The causal significant relation among the messages is defined by the compatibility relation and the information flow relation. In this paper, we would like to discuss what is the causally ordered delivery of message at the application level.

In section 2, we present the system model. In section 3, we discuss how to realize the application-oriented causal order.

2 System Model

2.1 System structure

The distributed computation is composed of computation of operations in the objects and communications among the objects. An object o is defined to be a pair of data D_o and a collection P_o of abstract operations for manipulating D_o. On receipt of a *request* of op in P_o, o computes op and sends back the *response*. op may change D_o. A *group* G is defined to be a collection of multiple objects $o_1, \ldots, o_n$ $(n \geq 2)$. The objects in G are cooperated with each other by sending requests and receiving responses.

2.2 Computation model

An object o_1 sends o_2 a *request* m_1 to compute an operation op_1. On receipt of m_1 sent by o_1, o_2 computes op_1 of m_1 if o_2 supports op_1. Here, o_1 and o_2 are the *sender* and *receiver* of m_1, respectively. On completion of op_1, o_2 sends the *response* message back to o_1. Thus, there are two kinds of messages, i.e. *requests* and *responses*. The request message m_1 sent by o_1 carries op_1 and input data in_1, i.e. $m_1 = \langle op_1, in_1 \rangle$ to o_2. If $in_1 \neq \phi$, information in o_1 is flown into o_2. The response m_2 of m_1 carries the output data of op_1, i.e. $m_2 = \langle out_2 \rangle$. If $out_2 \neq \phi$, information in o_2 is flown into o_1. If o_1 issues a request $m_3 = \langle op_3, in_3 \rangle$ 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.

2.3 Conflict operations

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

[Definition] For every state s of the object o, two operations op_1 and op_2 supported by o are *compatible* iff $op_1(op_2(s)) = op_2(op_1(s))$. □

Two operations op_1 and op_2 *conflict* iff they are not 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, op_1 can be computed.

Here, suppose that o_j sends $m_1 = \langle op_1, in_1 \rangle$ to o_i and o_k sends $m_2 = \langle op_2, in_2 \rangle$ to o_i. Suppose that o_i computes op_2 after op_1 completes. If $in_1 \neq \phi$ and op_1 changes the state of o_i, the information is flown from o_j to o_i. op_2 is computed and then o_i sends the response $m_3 = \langle out_2 \rangle$ to o_k. Here, if op_1 and op_2 conflict and $out_2 \neq \phi$, m_3 may include some information carried by m_1, i.e. m_1 *causally effects* m_3.

2.4 Instantiation of operation

Each time o receives op, a thread for computing op is created if op is compatible 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 object of op and the thread is removed. Each thread for op is computed sequentially, i.e. a sequence of *actions*. The action is an atomic primitive operation in the object. The computation of op is *instance* of op in o. The computation of op is considered to be *atomic* by the sender of op_i.

3 Message Precedence

3.1 Message types

If a message m carries *in* of the request and *out* of the response, information in the sender of m is flown into the receiver. Hence, it is important to consider whether the messages carry data or not. The second point is concerned with whether the operation changes the state of the object or not. The request $m = \langle op, in \rangle$ 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. The response $m = \langle out \rangle$ is typed as γ-response where γ is O if $out \neq \phi$, N otherwise.

3.2 Message precedence in object

3.2.1 Send-send precedence

For m_1 and m_2 sent by o_i, there are the following cases [Figure 1]:

C1. m_1 and m_2 are sent by the same instance op_1^i.

C2. m_1 and m_2 are sent by different instances op_1^i and op_2^i, respectively:

C2.1. $op_1^i \Rightarrow op_2^i$.

C2.2. op_1^i and op_2^i are interleaved.

In C1, m_1 *precedes* m_2 in o_i ($m_1 \prec_i m_2$). In C2, if m_1 and m_2 carry no data, $m_1 \parallel_i m_2$. 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 C2.1, 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 C2.1 where ○ means "$m_1 \prec_i m_2$" if op_1^i and op_2^i conflict and - means "$m_1 \parallel_i m_2$". In C2.2, $m_1 \parallel_i m_2$.

3.2.2 Receive-send precedence

Next, suppose that o_i sends m_2 after receiving m_1. There are the same cases as C1 and C2 [Figure 2]. If m_1 is the response without data, i.e. N-response, m_1

---

*グループ通信におけるオブジェクトに基づいた因果関係
†立川 敬行　滝沢 誠
‡東京電機大学

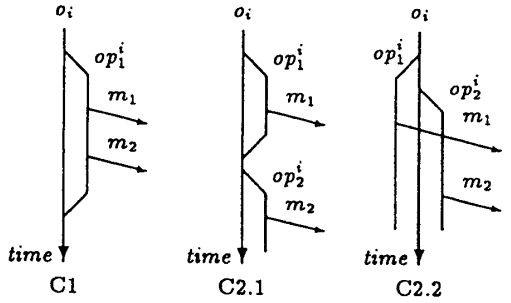

Figure 1: Send-send precedence

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

Table 1: Send - send (C2.1)

$\| m_2$. Here, suppose that m_1 is the request of op_1^i or O-response. In C1, if m_2 does not include data, $m_1 \|_i m_2$. Hence, $m_1 \prec_i m_2$ 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 "$\prec_i$" for C1. △ 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 C2.1, 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 "$\prec_i$" for C2.1 where op_1^i and op_2^i conflict. In C2.2, $m_1 \|_i m_2$.

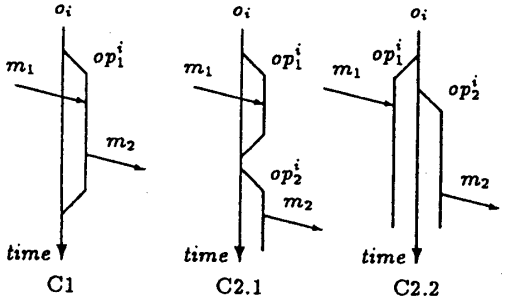

Figure 2: Receive-send precedence

3.2.3 Receive-receive precedence

Suppose that o_j sends m_2 after receiving m_1 and o_k receives m_1 and m_2. 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 :

M1. m_1 and m_2 are requests.
M2. m_1 is a request and m_2 is a response.
M3. m_1 is a response and m_2 is a request.
M4. m_1 and m_2 are responses.

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

For (a), only M2 and M4 can be considered. Here, $m_1 \prec_k m_2$.

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

Table 2: Receive - send (C1)

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

Table 3: Receive - send (C2.1)

Next, let us consider (b). In M1, m_1 is the request of op_1^k and m_2 is the request of op_2^k. If op_1^k and op_2^k conflict in o_k, $op_1^k \Rightarrow op_2^k$. Here, $m_1 \prec_k m_2$. Otherwise, $m_1 \|_k m_2$.

In M2, m_1 is a request of op_1^k. If op_1^k and op_2^k conflict in o_k, $op_1^k \Rightarrow op_2^k$. Hence, $m_1 \prec_k m_2$. If not conflict, $m_1 \|_k m_2$.

In M3, m_2 is a request of op_2^k. Like M2, $m_1 \prec_k m_2$ if op_1^k and op_2^k conflict. Otherwise, $m_1 \|_k m_2$.

In M4, 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, $m_1 \|_k m_2$. Suppose that op_1^k and op_2^k conflict in o_k. If m_1 and m_2 are N-responses, $m_1 \|_k m_2$. 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. In this case, op_2^k has to be aborted by the time out or deadlock resolution mechanism.

3.3 Message precedence among objects

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. In M1, suppose that m_1 and m_2 are requests of op_1 and op_2, respectively. The common destinations 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.

4 Concluding Remarks

In this paper, we have discussed how to support the causally ordered delivery of messages from the application point of view. Based on the compatibility relation among the operations, the significant causal order is decided.

References

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

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