4 F − 8

Protocol for A Group of Multiple Objects [*]

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

1 Introduction

Distributed applications are realized by a *group* of multiple application objects. Many papers have discussed so far how to support the causally ordered delivery of messages at the network level. An object o is an encapsulate of data and *abstract* operations for manipulating the data. On receipt of a *request* message with an operation op, o computes op and sends back a response message. States of the objects depend on in what order operations are computed. A *conflicting* relation among operations is defined for each object based on the semantics of the object. The ***significantly precedent relation*** among 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 ordered delivery of messages where only messages to be ordered are delivered to the application objects in the order. We propose an *object vector* to significantly order messages, which are based on the logical clocks of the objects.

In section 2, we discuss the significant precedency among messages. In section 3, the OG protocol is discussed.

2 Significantly Ordered Delivery

2.1 Object-based systems

A *group* G is a collection of objects $o_1, \ldots, o_n$ $(n \geq 1)$ which are cooperating by exchanging requests and responses through the network. We assume that the network is *less reliable* and *not synchronous*. Let $op(s)$ denote a state obtained by applying an operation op to a state s of o_i. Two operations op_1 and op_2 of an object o_i are *compatible* iff $op_1(op_2(s)) = op_2(op_1(s))$ for every state s of o_i. op_1 and op_2 *conflict* iff they are not compatible.

Each time o_i receives a request of op, a thread is created for op. The thread is referred to as an *instance* of op. op^i denotes an instance of op in o_i. Only if all the actions computed in op complete successfully, the instance of op *commits*. Otherwise, op *aborts*. That is, op is *atomically* computed. op_i may further invoke operations. Thus, the computation of op is *nested*.

2.2 Significant precedence

An operation instance op_1^i *precedes* op_2^i $(op_1^i \Rightarrow_i op_2^i)$ iff op_2^i is computed after op_1^i completes in o_i. op_1^i *precedes* op_2^j $(op_1^i \Rightarrow op_2^j)$ iff $op_1^i \Rightarrow_i op_2^j$ for $i = j$, op_1^i invokes op_2^j, or $op_1^i \Rightarrow op_3^k \Rightarrow op_2^j$ for some op_3^k. op_1^i and op_2^j are *concurrent* $(op_1^i \parallel op_2^j)$ iff neither $op_1^i \Rightarrow op_2^j$ nor $op_2^j \Rightarrow op_1^i$.

A message m_1 *causally precedes* m_2 if the sending event of m_1 precedes m_2[2]. Suppose an object o_i sends m_1 to o_j and o_k, and o_j sends m_2 to o_k after receiving m_1. Here, m_1 causally precedes m_2. Hence, o_k has to receive m_1 before m_2. We define a precedent relation "$\rightarrow$" among m_1 and m_2 which is significant for the application based on the concept of objects.

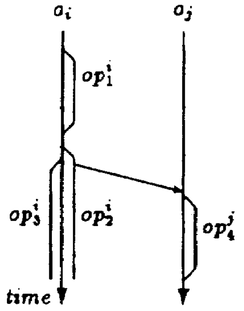

Figure 1: Precedence of operations

[Definition] A message 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
   a. m_1 and m_2 are sent by a same operation instance, or
   b. 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
   a. m_1 and m_2 are received and sent by the same operation instance, or
   b. 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 significantly precedes $m_2 (m_1 \rightarrow m_2)$. □

2.3 Ordered delivery

We define a significantly ordered delivery (SO) of message which order only significant messages.

[Significantly ordered delivery (SO)] A message m_1 is delivered before m_2 in a common destination o_i of m_1 and m_2 if the following condition holds :

- if $m_1 \rightarrow m_2$,
  - m_1 and m_2 are received by the same operation instance, or
  - an operation instance op_1^i receiving m_1 conflicts with op_2^i receiving m_2 in o_i and one of m_1 and m_2 is a request,
- if m_1 and m_2 are conflicting requests and $m_1 \parallel m_2$, m_1 is delivered before m_2 in another common destination of m_1 and m_2. □

3 Protocol

3.1 Object vector

The *vector clock* [?] $V = \langle V_1, \ldots, V_n \rangle$ is used to causally order messages received in most group protocols. Significant messages are defined in context of operation instances and in nested invocations. Hence, a group is considered to be composed of instances. In

---

[*] 複数オブジェクト間でのグループ通信プロトコル

[†] 榎戸 智也 桧垣 博章 滝沢 誠

[‡] 東京電機大学

the vector clock, the group has to be frequently resynchronized each time instances are initiated and terminated. In this paper, we propose an *object vector* to causally order only the significant messages.

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

- I1. If op^i_t starts after op^i_u starts in an object o_i, $t(op^i_t) > t(op^i_u)$.
- I2. If o_i initiates op^i_t after receiving a request op_t from op^j_u, $t(op^i_t) > t(op^j_u)$.

o_i manipulates a variable oid showing the linear clock[2] as follows : (1) Initially, $oid := 0$. (2) $oid := oid + 1$ if an instance op^i_t is initiated in o_i. (3) On receipt of a message from op^j_u, $oid := \max(oid, oid(op^j_u))$. When op^i_t is initiated in o_i, $oid(op^i_t) := oid$. Then, $t(op^i_t)$ is given a concatenation of $oid(op^i_t)$ and the object number $ono(o_i)$ of o_i. $t(op^i_t) > t(op^j_u)$ if 1) $oid(op^i_t) > oid(op^j_u)$ or 2) $oid(op^i_t) = oid(op^j_u)$ and $ono(o_i) > ono(o_j)$. It is clear that the instance identifiers satisfy I1 and I2.

Each action e in op^i_t is given an event number $no(e)$. o_i manipulates a variable no_i to give the event number to each event e in o_i as follows : (1) Initially, $no_i := 0$. (2) $no_i := no_i + 1$ if e is a sending action. $no(e) := no_i$; That is, the event number is incremented by one each time a sending event occurs. Each action e in op^i_t is given a global event number $tno(e)$ as the concatenation of $t(op^i_t)$ and $no(e)$.

o_i has a vector of variables $V^i = \langle V^i_1, \ldots, V^i_n \rangle$. Each V^i_j is initially 0. Each time an instance op^i_t is initiated in o_i, op^i_t is given a vector $V^i_t = \langle V^i_{t1}, \ldots, V^i_{tn} \rangle$ where $V^i_{tj} := V^i_j$ for $j = 1, \ldots, n$. op^i_t manipulates V^i_t as follows : (1) If op^i_t sends a message m, m carries the vector V^i_t as $m.V$ where $m.V_j := V^i_{tj}$ for $j = 1, \ldots, n$ $(j \neq i)$, $no_i := no_i + 1$, and $V^i_{ti} := no_i$; (2) If op^i_t receives a message m from o_j, $V^i_{tj} := m.V_j$; (3) If op^i_t commits, $V^i_j := \max(V^i_j, V^i_{tj})$ for $j = 1, \ldots, n$;

3.2 Message transmission and receipt

A message m includes the following fields : (1) $m.src$ = sender object of m. (2) $m.dst$ = set of destination objects. (3) $m.type$ = message type, i.e. *request*, *responce*, *commit*, *abort*. (4) $m.op$ = operation. (5) $m.tno$ = global event number $\langle m.t, m.no \rangle$, i.e. $tno(m)$. (6) $m.V$ = *object vector* $\langle V_1, \ldots, V_n \rangle$. (7) $m.SQ$ = vector of sequence numbers $\langle sq_1, \ldots, sq_n \rangle$. (8) $m.d$ = data.

An object o_i manipulates variables $sq_1, \ldots, sq_n$ to detect a message gap, i.e. messages lost or unexpectedly delayed. Each time o_i sends a message to o_j, sq_j is incremented by one. Then, o_i sends a message 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. o_j *correctly* receives m 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.

Suppose op^i_t sends a request op^j. o_i constructs a message m as follows : (1) $m.src := o_i$; (2) $m.dst :=$ set of destinations; (3) $m.type := request$; (4) $m.op := op^j$; (5) $m.tno = \langle m.t, m.no \rangle = \langle t(op^i_t), no_i \rangle$; (6) $m.V_j := V^i_{tj}$ for $j = 1, \ldots, n$; (7) $sq_j := sq_j + 1$ for every o_j in $m.dst$; (8) $m.sq_j := sq_j$ for $j = 1, \ldots, n$;

3.3 Message delivery

A *receipt* vector $RV = \langle RV_1, \ldots, RV_n \rangle$ is given to each message m received from o_i. $m.RV$ is manipulated as follows :

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

$m.RV$ is the same as $m.V$ except $m.RV_i$ for an object o_i which sends m. Messages m_1 and m_2 received an object are ordered by the following rule.

[Ordering rule] A message m_1 precedes m_2 $(m_1 \Rightarrow m_2)$ if the following condition holds :

If $m_1.V < m_2.V$ and $m_1.RV < m_2.RV$,

- $m_1.op = m_2.op$, or
- $m_1.op$ conflicts with $m_2.op$.

else $m_1.type = m_2.type = request$, $m_1.op$ conflicts with $m_2.op$, and $m_1.tno < m_2.tno$. □

It is clear that the following theorem to hold.

[Theorem] m_1 significantly precedes m_2 $(m_1 \rightarrow m_2)$ iff $m_1 \Rightarrow m_2$. □

The messages in RQ_i are ordered in $\Rightarrow$. Messages not ordered in $\Rightarrow$ are stored in the receipt order.

[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 $m \rightarrow m_1$. □

[Definition] A message m in RQ_j is *ready* if operation conflicting with $m.op$ is not computed in o_j. □

In addition, only significant messages in RQ_j are delivered by the following procedure.

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

[Theorem] The OG protocol delivers a message m_1 before m_2 if m_1 significantly precedes m_2. □

If 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 data to o_j for some predetermined δ time units. δ is proportional to delay time between o_i and o_j.

4 Concluding Remarks

In this paper, we have discussed how to support the *significantly* ordered delivery of messages from the application point of view. Based on the conflicting relation among abstract operations, we have defined the significantly precedent relation among request and response messages. We have discussed the object vector to significantly order messages in the object-based systems. The size of the object vector depends on the number of objects, not the number of operation instances.

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