Group Protocol for Multiple Objects

Tomoya Enokido, Hiroaki Higaki, and Makoto Takizawa
Dept. of Computers and Systems Engineering
Tokyo Denki University
Email {eno, hig, taki}@takilab.k.dendai.ac.jp

Abstract

Distributed applications are realized by cooperation of multiple objects. A state of the object depends on in what order the object exchanges request and response messages. In this paper, we newly define a significantly precedent order of messages based on a conflicting relation among requests. The objects can be mutually consistent if the objects take messages in the significantly precedent order. We discuss a protocol which supports the significantly ordered delivery of request and response messages. Here, an object vector is newly proposed to significantly order messages.

1 Introduction

Distributed applications are realized by a *group* of multiple application objects. Many papers [3, 10] discussed how to support the causally ordered delivery of messages at the network level in presence of message loss and stop faults of the objects. Cheriton *et al.* [4] point out that it is meaningless at the application level to causally order all messages transmitted in the network. Ravindran *et al.* [11] discuss how to support the ordered delivery of messages based on the message precedency explicitly specified by the application. Agrawal *et al.* [8] define *significant* messages which change the state of the object. Raynal *et al.* [1] discuss a group protocol for replicas of file where write–write semantics of messages are considered. The authors [5] discuss a group protocol for replicas where a group is composed of transactions issuing read and write requests to the replicas.

An object o is encapsulation of data and methods. On receipt of a *request* message with a method op, the object o computes op and sends back a response message with the result of op. Here, the method op may further invoke another method, i.e. *nested* invocation. States of the objects depend on in what order methods are computed. A *conflicting* relation among methods is defined for each object based on the semantics of the object. If a pair of methods sending and receiving messages conflict in an object, the messages have to be received in the computation order of the methods. Thus, 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 at the application level are delivered to the application objects in the order. Takizawa *et al.* [12] show a protocol for a group of objects, which uses the real time clock. However, it is not easy to synchronize real time clocks in distributed objects. We newly propose an *object vector* to significantly order messages.

In section 2, we discuss the significant precedency among messages. In section 3, the OG protocol is discussed. In section 4, we present the implementation and evaluation of the OG protocol.

2 Significantly Ordered Delivery in Object-based Systems

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 response messages in the network. We assume that 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.

An object o_i can be manipulated only through methods supported by o_i. Let $op(s)$ denote a state obtained by applying a method op to a state s of the object o_i. A pair of methods op_1 and op_2 of 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. The *conflicting* relation C_i among the methods is specified when o_i is defined. We assume that is symmetric but not transitive. A pair of request messages m_1 of a method op_1 and m_2 of op_2 *conflict* iff op_1 and op_2 conflict. Suppose op_1 is issued to o_i. If op_1 conflicts with some method being computed in o_i, op_1 has to wait until op_2 completes.

Each time an object o_i receives a request message of a method op, a thread is created for op. The thread is as an *instance* of op in o_i, which is denoted by op^i. Only if all the actions computed in op complete successfully, i.e. *commit*, the instance of op commits. Otherwise, op *aborts*. op may further invoke methods of other objects. Thus, the invocation is *nested*.

2.2 Significant precedence

A method instance op_1^i *precedes* another one op_2^i ($op_1^i \Rightarrow_i op_2^i$) iff op_2^i is started to be 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 $j = i$, 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* another one m_2 if the sending event of m_1 precedes m_2 [3,7]. Suppose an object o_i sends a message m_1 to objects 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 ***significantly precedent*** relation "$\rightarrow$" among messages m_1 and m_2, which is significant for applications in the object-based system. There are the following cases :

S. An object o_i sends m_2 after m_1 [Figure 1].

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

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

R. o_i sends m_2 after receiving m_1 [Figure 2].

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. $op_1^i \parallel op_2^i$.

We discuss how messages are significantly preceded for each of the cases. First, let us consider the case S [Figure 1] where an object o_i sends a message m_1 before m_2. In S1, m_1 *significantly precedes* m_2 ($m_1 \rightarrow m_2$) since m_1 and m_2 are sent by the same instance 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 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 $m_2 \rightarrow m_1$. Here, m_1 and m_2 are ***significantly concurrent*** ($m_1 \parallel m_2$). Suppose op_1^i and op_2^i conflict. The output data carried by the messages m_1 and m_2 in "$op_2^i \Rightarrow op_1^i$" 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. $m_1 \rightarrow m_2$. In S2.2, $op_1^i \parallel op_2^i$. Since op_1^i and op_2^i are not related, $m_1 \parallel m_2$.

In the case R [Figure 2], o_i sends m_2 after receiving m_1. 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 a method invoked by op_1^i. m_2 is the response of op_1^i or a request of a method invoked by op_1^i. The output 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] A message m_1 ***significantly precedes*** another message 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 method instance, or
   b. a method sending m_1 conflicts with a method 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 a same method instance, or
   b. a method receiving m_1 conflicts with a method sending m_2.
3. $m_1 \rightarrow m_3 \rightarrow m_2$ for some message m_3. □

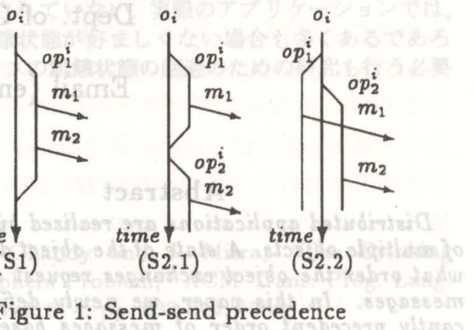

Figure 1: Send-send precedence

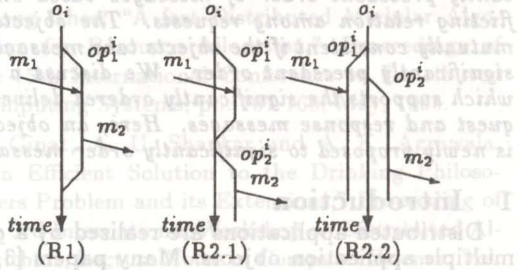

Figure 2: Receive-send precedence

[Proposition] A message m_1 causally precedes a message m_2 if m_1 significantly precedes m_2 ($m_1 \rightarrow m_2$). A message m is significantly preceded by only messages related with m.

2.3 Ordered delivery

Suppose an object o_k sends a message m_1 to two objects o_i and o_j, and o_k sends m_2 to o_h, o_i, and o_j [Figure 3]. 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.

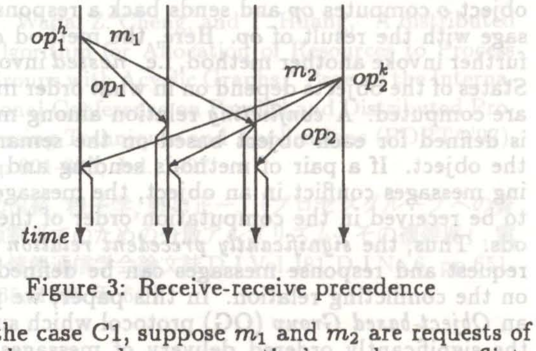

Figure 3: Receive-receive precedence

In the case C1, suppose m_1 and m_2 are requests of methods op_1 and op_2, respectively, and op_1 conflicts with op_2 in the objects o_i and o_j. If $m_1 \parallel m_2$, m_1 and m_2 may be delivered in o_i and o_j in different orders. However, the state of o_i obtained by computing op_1

and op_2 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 mutually consistent, m_1 and m_2 have to be delivered to o_i and o_j in the same order. Thus, a pair of requests m_1 and m_2 have to be delivered in every pair o_i and o_j of common destinations in the same order if the requests m_1 and m_2 conflict in o_i and o_j. In C2 and C3, m_1 and m_2 can be delivered in any order.

Suppose o_i receives messages m_1 and m_2. First, suppose $m_1 \parallel m_2$. If m_1 and m_2 are requests sent to one object o_i, o_i can receive m_1 and m_2 in any order. Otherwise, the cases C1, C2, and C3 are adopted. Next, suppose m_1 *significantly precedes* m_2 ($m_1 \rightarrow m_2$). There are the following cases :

T. o_i receives m_2 before m_1 [Figure 4].

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_2^i precedes op_1^i ($op_2^i \Rightarrow op_1^i$).

  T2.2. op_1^i and op_2^i are concurrent.

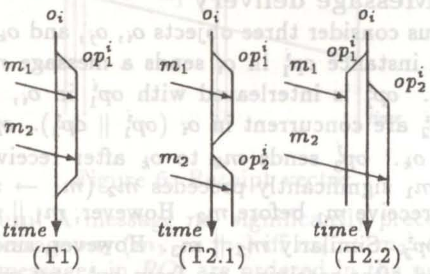

Figure 4: Receive-receive precedence

In T1, m_1 has to be delivered to the object o_i before m_2 since m_1 significantly precedes m_2 ($m_1 \rightarrow m_2$). In T2, m_1 and m_2 are received by different instances op_1^i and op_2^i. If op_1^i and op_2^i are concurrent ($op_1^i \parallel op_2^i$) 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. Suppose m_3 is sent to op_1^i and m_4 to op_2^i and $m_4 \rightarrow m_3$. Even if $op_1^i \Rightarrow op_2^i$ and m_1 is delivered before m_2, deadlock occurs because $m_4 \rightarrow m_3$. Thus, 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)] A message m_1 is delivered before another message m_2 in a common destination o_i of m_1 and m_2 if the following condition holds :

- if $m_1 \rightarrow m_2$,
  - a same instance receives m_1 and m_2, or
  - a method 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. □

[Theorem] No communication deadlock occurs if every message is delivered by the SO rule. □

The system is consistent if every message is delivered by the SO rule.

3 Object-Based Group Protocol

3.1 Object vector

The *vector clock* [9] $V = \langle V_1, \ldots, V_n \rangle$ is widely used to causally order messages in most group protocols. Each object o_i manipulates a vector clock $V = \langle V_1, \ldots, V_n \rangle$ ($i = 1, \ldots, n$). Each element V_i 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 a message m', o_i changes V as $V_j := \max(V_j, m'.V_j)$ for $j = 1, \ldots, n$ and $j \neq i$. A message m_1 causally precedes another message m_2 iff $m_1.V < m_2.V$.

The significant precedency of messages is defined in context of instances invoked and in nested invocations while the causality is defined for messages sent and received by "objects". Hence, a group is considered to be composed of method instances, not objects. In the vector clock, the group has to be frequently resynchronized [3,4,7–9,12] each time instances are initiated and terminated. In this paper, we newly propose an *object vector* to causally order only the significant messages.

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

I1. If op_t^i starts after op_u^i starts in an object o_i, $id(op_t^i) > id(op_u^i)$.

I2. If o_i initiates op_t^i after receiving a request op_t from op_u^j, $id(op_t^i) > id(op_u^j)$.

The object o_i manipulates a variable oid, initially 0, showing the linear clock [7] as follows :

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

When an instance op_t^i is initiated in the object o_i, the instance identifier $id(op_t^i)$ is given a concatenation of oid and the object number $ono(o_i)$ of o_i. Here, let $oid(op_t^i)$ show oid of $id(op_t^i)$. $id(op_t^i) > id(op_u^j)$ if 1) $oid(op_t^i) > oid(op_u^j)$ or 2) $oid(op_t^i) = oid(op_u^j)$ and $ono(o_i) > ono(o_j)$. It is clear that the instance identifiers satisfy I1 and I2.

Each action e in op_t^i is given an event number $no(e)$. o_i manipulates a variable no_i for each action e, i.e. $no(e) := no_i$ in o_i as follows :

- Initially, $no_i := 0$.
- $no_i := no_i + 1$ if e is a sending action.

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

An object o_i manipulates a vector $V^i = \langle V_1^i, \ldots, V_n^i \rangle$. Each element V_j^i is initially 0. Each time an instance op_t^i is initiated on o_i, op_t^i is given $V_t^i = \langle V_{t1}^i,$

..., $V^i_{in}\rangle$ where $V^i_{tj} := V^i_j$ for $j = 1, \ldots, n$. Each element V^i_t is manipulated for op^i_t as follows :

- [op^i_t sends a message m] $no_i := no_i + 1$; $V^i_{ti} := \langle id(op^i_t),\ no_i\rangle$; m carries the vector V^i_t as $m.V$ where $m.V_j := V^i_{tj}$ ($j = 1, \ldots, n$).
- [op^i_t receives a message m from o_j] $V^i_{tj} := m.V_j$;
- [op^i_t commits] $V^i_j := \max(V^i_j, V^i_{tj})$ ($j = 1, \ldots, n$);
- [op^i_t aborts] V^i is not changed.

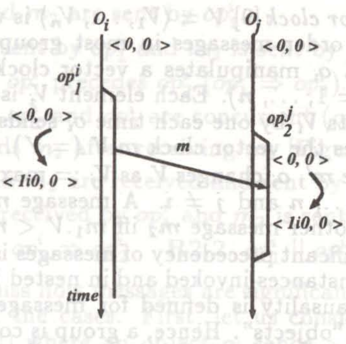

Figure 5: Object vector

In Figure 5, the vectors V^i and V^j are initially $\langle 0, 0\rangle$. An instance op^i_1 is initiated in o_i where $V^i_1 = \langle 0, 0\rangle$. After sending a message m to op^j_2, e.g. m is a request of op_2 to o_j, V^i_1 is changed to $\langle 1i0,\ 0\rangle$ where $1i0$ is the global event number of the sending action of m. m carries $V^i_1 (= \langle 1i0,\ 0\rangle)$ to op^j_2. On receipt of m, op^j_2 changes V^j_2 to $\langle 1i0,\ 0\rangle$. After op^j_2 commits, V_j of o_j is changed to be $\langle 1i0,\ 0\rangle$.

3.2 Message transmission and receipt

A message m includes the following fields:

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

If m is a request message, $m.tno$ is a global event number of the sending action of m. $m.id$ shows the instance identifier and $m.no$ indicates the event number in the instance. If m is a response message of a request m', $m.tno = m'.tno$ and $m.op = m'.op$.

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 another object o_j, sq_j is incremented by one. Then, o_i sends a message m to every destination object in $m.dst$. The object o_j can detect a gap between messages received from o_i by checking the sequence number. o_j manipulates variables $rsq_1, \ldots, rsq_n$ to receive messages. rsq_j shows a sequence number of message which o_i expects to receive next from o_j. On receipt of m from o_i, there is no gap 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 is sent by o_i. 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. The selective retransmission to recover from the message loss is used in the protocol. 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. o_j enqueues m in a receipt queue RQ_j even if a gap is detected on receipt of m.

Suppose an instance op^i_t in an object o_i invokes a method op. Here, op may be sent to multiple objects. o_i constructs a message m for op as follows and sends m to the destination objects :

$m.src := o_i$; $m.dst :=$ set of destinations;
$m.type :=$ *request*; $m.op := op$;
$m.tno = \langle m.id,\ m.no\rangle := \langle id(op^i_t),\ no_i\rangle$);
$sq_h := sq_h + 1$ for every o_h in $m.dst$;
$m.V_j := V^i_{tj}$ and $m.sq_j := sq_j$ for $j = 1, \ldots, n$;

3.3 Message delivery

Let us consider three objects o_i, o_j, and o_k [Figure 6]. An instance op^i_1 in o_i sends a message m_1 to o_j and o_k. op^i_2 is interleaved with op^i_1 in o_i, i.e. op^i_1 and op^i_2 are concurrent in o_i ($op^i_1 \parallel op^i_2$). op^i_2 sends m_3 to o_k. op^j_3 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, $m_1 \parallel m_3$ since $op^i_1 \parallel op^i_2$. Similarly $m_2 \parallel m_3$. However, since op^j_3 is initiated after receiving m_1 from op^i_1 and $op^i_1 \parallel op^i_2$, $m_1.V = m_3.V$. Hence, $m_2.V > m_3.V$. Although o_k can receive m_2 and m_3 in any order since $m_2 \parallel m_3$, "m_2 precedes m_3" by the object vector. In order to resolve this problem, an additional *receipt vector* $RV = \langle RV_1, \ldots, RV_n\rangle$ is given to each message m received from o_i. $m.RV$ shows RV in m. $m.RV$ is the same as $m.V$ except that $m.RV_i$ shows the global event number of the sending event of m for an object o_i which sends m. $m.RV$ is manipulated as follows :

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

In Figure 6, $id(op^i_1) < id(op^i_2)$ because op^i_2 starts after op^i_1. Hence, $m_1.RV < m_3.RV$ as shown in Table 1. The instance op^i_1 sends a message m_1 to objects o_j and o_k where $m.tno = 1i0$ and $m.V = \langle 0,\ 0,\ 0\rangle$. On receipt of m_1, o_j enqueues m_1 into a receipt queue RQ_j. Here, o_j gives RV to m_1, i.e. $m_1.RV = \langle 1i0, 0, 0\rangle$ while $m_1.V$ is still $\langle 0, 0, 0\rangle$. Table 1 shows values of tno, V, and RV. $m_1.V < m_2.V$ and $m_1.RV < m_2.RV$. On the other hand, $m_2.V > m_3.V$ but $m_2.RV$ and $m_3.RV$ are not comparable.

Following this example, a pair of messages m_1 and m_2 are ordered by the following rule.

[Ordering rule] A message m_1 precedes another one m_2 ($m_1 \Rightarrow m_2$) if the following one 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 $m_1.tno < m_2.tno$. □

In Figure 6, $m_1 \Rightarrow m_2$ since $m_1.V < m_2.V$ and $m_1.RV < m_2.RV$. On the other hand, $m_1.V = m_3.V$ but $m_1.RV < m_3.RV$. Accordingly, $m_1.op$ and $m_3.op$ are checked. Since op_1^i and op_2^i are compatible, m_1 and m_3 are not ordered in the precedent relation "$\Rightarrow$".

Table 1: Object vectors

m	$m.tno$	$m.V$	$m.RV$
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	$2i0$	$\langle 0, 0, 0\rangle$	$\langle 2i0, 0, 0\rangle$

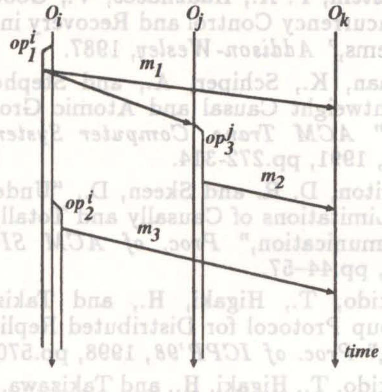

Figure 6: Receipt vector.

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

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

[Stable message] A message m which an object o_i sends to o_j and is stored in the receipt queue RQ_j 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, where $m \rightarrow m_1$. □

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. A message m in RQ_j is *ready* in an object o_j if no method conflicting with the method $m.op$ 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 m 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 $m_1 \rightarrow m_2$. □

If an object o_i sends no message to another object 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. 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 the receipt confirmation of m from o_j in 2δ after o_i sends m to o_j. Here, o_i resends m.

4 Implementation and Evaluation

4.1 Implementation

An OG protocol module is implemented as a process of Solaris 2.6 in the Sun workstation. Each processor has one OG protocol module and objects. The OG modules exchange messages by using UDP [15]. The OG module in each processor delivers messages to the objects in the significantly precedent order. A transaction in a client processor issues request messages to objects in server processors. Each OG protocol module in a processor p_t includes two threads, *Rec* for receiving messages and *Snd* for sending messages [Figure 7]. These threads share the variables showing the sequence numbers sq, rsq, the object vector V, the event number no, and the instance identifier id in the shared memory. The *Rec* and *Snd* threads mutually exclusively manipulate the variables by using the semaphore. The OG module delivers messages in the delivery queue DQ of each object in the significantly precedent order by the ordering rule.

Each object o_i is realized by one process. The object o_i takes a top message in the delivery queue DQ. On taking a request op_t from DQ, o_i is locked in a mode $\mu(op_t)$. If o_i could be locked, a thread for op_t is created. Otherwise, op_t blocks in a block queue of o_i. In this implementation, unless an object could be locked by a transaction in a fixed time after the lock request is issued, the transaction aborts. In this implementation, the *semi-open* locking scheme is adopted to release objects locked. Suppose that the method op_t of o_i invokes methods $op_{t1}, \ldots, op_{th_t}$ on objects $o_{i1}, \ldots, o_{ih_t}$ ($h_t \geq 1$). Before computing op_{tu}, the object o_{iu} is locked. If op_t commits, the objects $o_{i1}, \ldots, o_{ih_t}$ are released while o_i is still being locked. If op_t aborts, not only $o_{i1}, \ldots, o_{ih_t}$ but also o_i are released. The object o_i is released if the method invoking op_t completes or op_t aborts.

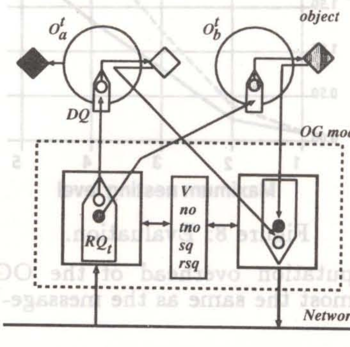

Figure 7: Implementation of OG protocol.

4.2 Evaluation

In the evaluation, each processor is implemented in one Ultra Sparc CPU in a Cray Super Server 6400 with 10 CPUs. Three objects x, y, and z are distributed in the processors. Each of x and y supports three types of methods and z supports two types of methods. Each method invokes one or two methods in other objects. Each processor has one object. First, eight transactions are sequentially initiated in each processor. Each transaction invokes one methods randomly selected from eight methods supported by the objects x, y, and z. A method invoked by the transaction furthermore invokes other methods. Each transaction randomly invokes one method in the system. Then, the method invokes other methods. In the evaluation, each transaction invokes methods in a nested manner at a fixed number of levels. Table 2 shows number of transactions issued for each nesting level. We measure the total response time of the transactions in the OG protocol and the message-based protocol. The average response time is calculated from the response time obtained by computing four times the evaluation. Figure 8 shows the average response time of the transactions for the level of nested invocation. The dotted line shows the response time of the message-based protocol. The straight line indicates the OG protocol. The figure shows the transactions can finish earlier than the message-based one because insignificant request messages are computed without waiting for messages causally preceding in the OG protocol.

Table 2: Number of transactions

Nesting level	1	2	3	4	5
Number of transactions	8	14	25	39	42

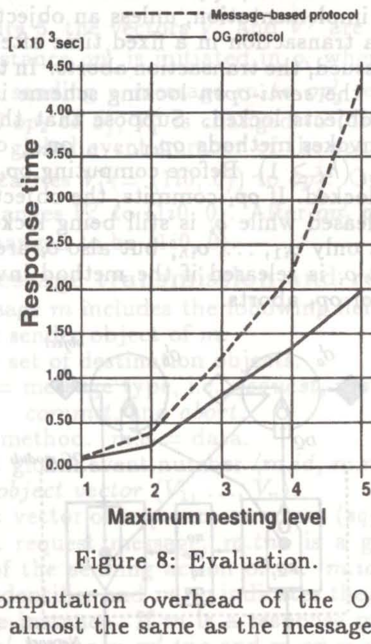

Figure 8: Evaluation.

The computation overhead of the OG protocol module is almost the same as the message-based protocol.

5 Concluding Remarks

In this paper, we have discussed how to support the *significantly* ordered delivery of messages. While network messages are causally ordered in most group protocols, only messages to be causally ordered at the application level are ordered. The system is modeled to be a collection of objects. Based on the conflicting relation among methods, 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 method instances. We have presented the implementation of the OG protocol and how the OG protocol reduces the response time of the transactions through the evaluation.

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. Computer Systems*, Vol.9, No.3, 1991, pp.272-314.

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

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

[8] 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.

[9] 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.

[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 Realtime Applications," *Proc. of IEEE ICDCS-18*, 1998, pp.40–47.

[14] 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.

[15] User Datagram Protocol, RFC 0768, 1980, pp. 1–3.