Distributed Protocol for Selective Intra-Group Communication

Takayuki Tachikawa and Makoto Takizawa

Dept. of Computers and Systems Engineering
Tokyo Denki University
Ishizaka, Hatoyama, Hiki-gun, Saitama 350-03, JAPAN
E-mail {tachi, taki}@takilab.k.dendai.ac.jp

Abstract

In distributed applications, a group of application processes is established and the processes in the group communicate with one another, i.e. intra-group communication. Here, messages have to be reliably and causally delivered to all the destinations. In addition, the processes send messages to any subset of the group at any time. This paper presents an intra-group communication protocol which provides the group of application processes with the selective and causally ordered (SCO) delivery of messages. The SCO protocol is based on the fully distributed control scheme, i.e. no master controller, and uses the high-speed one-to-one network where messages may be lost due to the buffer overrun and congestion.

1 Introduction

Distributed applications like groupware [5] require group communications among multiple application processes. There are two kinds of group communications. The first type [17, 18, 20] is *intra-group* communication where after a group of processes is established, the processes communicate with one another in the group. This is the extension of the conventional *connection* concept among two processes to multiple processes. The processes might like to send messages to some processes, not necessarily all in the group. In the *selective* group communication [20], each process can send messages to any subset of the group at any time. [16] discusses a *selective sending-order preserving* (*SOP*) protocol where each process can receive messages destined to the process in the sending order. [20] presents a *selective totally ordering* (*STO*) protocol where every two common destination processes of every two messages receive the messages in the same order. ISIS [2, 3] supports the second type named *multicast* communication, where processes can send messages to a group of processes where every process can receive the messages in the causal order. [11, 19] discuss how to deliver messages in the causal order specified at the application level.

In the distributed applications, a group of application processes have to receive messages in the *causal* order [2, 3, 13, 18, 24]. In this paper, we would like to discuss an intra-group communication protocol which supports a group of application processes with the selective and causally ordered (*SCO*) delivery of messages. While [17, 18, 20] use the broadcast network and [2, 3] use the reliable one-to-one network, the SCO protocol uses the high-speed one-to-one network like ATM [1]. In the high-speed network, each process may fail to receive messages due to the buffer overrun because the transmission speed of the network is faster than the processing speed of the process, and messages may be lost due to the congestion. In order to take advantage of the high-speed network, the communication protocols are required to be *light-weighted*. We approach that the communication system supports the application process with SCO delivery by recovering from the message loss and out-of-order delivery of messages by directly using the high-speed network without assuming that there exists one lower layer supporting the non-loss and ordered delivery on the network. According to advances of VLSI technologies, each process can be considered to be reliable in the distributed applications like groupware. Therefore, we can assume that the processes are reliable but messages may be lost. While [3, 12, 14, 15] discuss how to treat the process faults assuming that the network is reliable.

The sender of each message m plays a role of controller in ISIS [2, 3] and Delta-4 [24], and there is one controller named *sequencer* in Amoeba [9]. The reliable delivery of m means that m is received by every destination process of m. These are non-distributed approaches based on the two-phase commitment [7]. The SCO protocol adopts the *distributed* approach where each process has to send other processes the acceptance confirmation of messages received while the process sends the acceptance confirmation to only the controller in the non-distributed approach. That is, more messages are transmitted in the distributed approach than the non-distributed one. [17, 18, 20] show that the distributed protocols have $O(n)$ message overhead in the broadcast network and $O(n^2)$ in the one-to-one network for number n of processes in the group. In this paper, we would like to discuss how to reduce the number of messages in the distributed control.

0-8186-7216-1/95 $04.00

That is, the acceptance confirmation is included in messages, i.e. *piggy back*. In addition, the process does not send the confirmation of messages as soon as the messages are received, i.e. *deferred confirmation*.

In section 2, the basic concepts are defined. In section 3, we discuss the data transmission procedure of the SCO protocol. In section 4, we evaluate the performance of the SCO protocol by comparing with the non-distributed protocols.

2 Basic Concepts

2.1 Layered structure

A communication system is composed of *application*, *system*, and *network* layers [Figure 1]. Each layer supports the higher layer with communication service through *service access points* (*SAPs*). The network layer provides the system layer with high-speed data transmission [1]. The messages may be lost in the high-speed network [1] due to the buffer overruns and congestions. A group $\mathcal{G}$ [21] of application processes $A_1, \ldots, A_n$ is supported by system processes $S_1, \ldots, S_n$, written as $\mathcal{G} = \langle S_1, \ldots, S_n \rangle$ $(n \geq 2)$. $S_1, \ldots, S_n$ cooperate to support the group communication service by using the network. After $\mathcal{G}$ is established, each A_i communicates with $A_1, \ldots, A_n$ in $\mathcal{G}$.

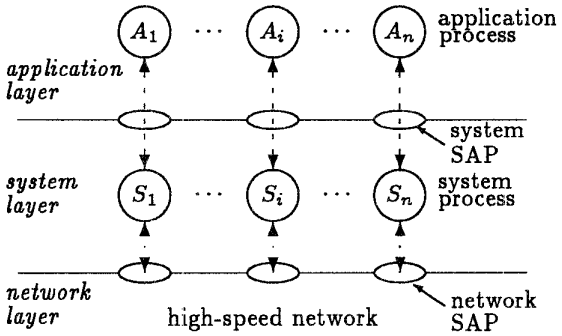

Figure 1: System model

2.2 Ordered delivery

Processes P_1, ..., P_n at each layer use service provided by the underlying layer as presented in 2.1. It is important to consider in what order the underlying layer delivers messages to the processes. The causal precedence relation "$\prec$" [3] among the messages is defined based on the Lamport's *happened-before* relation [2,10].

[Definition] For every pair of messages m and m', m *causally precedes* m' $(m \prec m')$ iff

(1) a process P_i sends m before m', or
(2) P_i sends m' after receiving m, or
(3) for some message m'', $m \prec m''$ and $m'' \prec m'$ □

m and m' are *causally coincident* $(m \parallel m')$ iff neither $m \prec m'$ nor $m' \prec m$. $m \preceq m'$ iff $m \prec m'$ or $m \parallel m'$.

[Definition]

(1) The underlying layer supports *selective information-preserved* delivery iff all messages are delivered to the destinations.
(2) The underlying layer supports *causally ordered* delivery iff for every pair of messages m and m', every P_i receives m before m' if $m \prec m'$. □

Figure 2 shows the data transmission among four processes P_1, P_2, P_3, and P_4. $m_1 \prec m_2 \prec m_3 \prec m_4$ because P_1 sends m_2 after m_1, P_2 sends m_3 after receiving m_2, and P_3 sends m_4 after receiving m_3. P_4 receives m_4 after m_2 if the causally ordered delivery is supported.

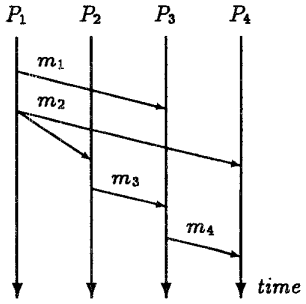

Figure 2: Causally ordered delivery

[Definition] The underlying layer supports *selective causally ordered* (*SCO*) delivery if it is selectively information-preserved and causally ordered. □

The high-speed one-to-one network does not support the SCO delivery, i.e. the messages may be lost or not be causally delivered. In this paper, we would like to discuss how to support a group of application processes with the SCO service by using the high-speed one-to-one network.

2.3 Acceptance levels

The system processes send and receive messages by using the network. We would like to discuss how a system process S_i accepts a message m received from S_j in the presence of message loss.

- S_i *simply* accepts m iff S_i takes m on receipt of m if m is destined to S_i.
- S_i *continuously* accepts m iff S_i simply accepts m and all the messages which S_j sends before m.
- S_i *causally* accepts m iff S_i simply accepts m and all the messages causally preceding m.

Unless S_i simply accepts m destined to S_i, S_i *loses* m. If S_i losses a message m' which S_j sends before m, S_i does not continuously accept m while S_i simply accepts m. The *gap* in a message sequence, i.e. message loss, can be easily detected by giving a unique sequence number to each message.

[Theorem 1] S_i does not causally accept a message m unless for every process S_h, there is a message m' causally following m, i.e. $m \prec m'$, which S_i continuously accepts from S_h. □

This theorem means that S_i can causally accept m after continuously accepting at least one message m' from every process, where $m \preceq m'$.

- S_i *reliably* accepts m iff S_i knows that every destination of m simply accepts m.

The continuously causal acceptance of m means that m is *stably* delivered without *gap* [13]. We assume that each message m' sent by S_h includes the acceptance confirmation of m which S_h has simply accepted before sending m'. Here, m' is referred to as *confirm* m. If S_i simply accepts messages confirming m from every destination of m, S_i reliably accepts m.

- S_i *continuously reliably* accepts m iff S_i reliably accepts m and all the messages which S_j sends m.
- S_i *causally reliably* accepts m iff S_i reliably accepts m and all the messages causally preceding m.

It is straightforward that S_i continuously reliably accepts m if S_i causally reliably accepts m. Here, m is referred to as *fully* accepted by S_i if S_i could deliver m to the application. In the SCO service, S_i fully accepts m if S_i causally reliably accepts m. Figure 3 shows the implication relation among the acceptance levels where $\alpha \rightarrow \beta$ shows that α implies β.

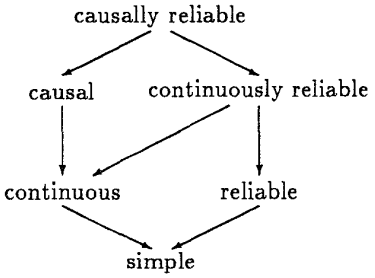

Figure 3: Acceptance levels

S_i has to causally accept m in the presence of loss of messages causally preceding m. S_i knows the loss of m' which S_j sends before m if S_i receives messages which S_j sends after m'. By S_j's retransmitting m' to S_i, S_i can continuously accept m.

2.4 Control schemes

There are three kinds of control schemes, i.e. *centralized*, *decentralized*, and *distributed* ones on how the system processes S_1, ..., S_n reliably accept a message m in the presence of message loss. Let r denote the average number of destinations of message. In the centralized protocol [9], there is one controller. In the decentralized one [2,3,24], a sender of m plays a role of the controller. They are based on the *two-phase commitment* protocol [7]. In Figure 4(1), S_1 plays a role of controller and sends a message m_1 to S_2 and S_3. S_2 and S_3 send m_2 and m_3 confirming m_1 to S_1. Then, S_1 sends m_4 to S_2 and S_3. Here, S_2 and S_3 reliably accept m_1. Totally $3r$ messages are transmitted and it takes three rounds.

In the distributed protocol, if every S_i simply accepts a message m from S_j, S_i sends messages confirming m to S_j and all the destinations of m. If S_i simply accepts the messages confirming m from all the destinations of m, S_i knows that m has been accepted by every destination of m, i.e. S_i reliably accepts m. Even if some S_k loses messages confirming m, S_k can ask another process if m is reliably accepted. If processes are not reliable, the three-phase protocol [16–18, 20–23] is required to support the reliable acceptance, i.e. m is accepted only if all the destinations simply accept m. Figure 4(2) shows the distributed protocol. After simply accepting p, S_2 and S_3 send the confirmation of m to the sender S_1 and the other destinations. Here, totally r^2 messages are transmitted and it takes two rounds.

Thus, the distributed approach requires more messages and less number of rounds than the centralized one. In order to decrease messages in the distributed one without increasing the number of rounds, we adopt the following strategies:

(1) the acceptance confirmation of messages accepted is carried back by the message sent, and
(2) each S_i does not send the acceptance confirmation as soon as S_i simply accepts.

Here, messages with and without data are referred to as *data* and *control* ones, respectively. After simply accepting a data message m from S_j, S_i sends a data message m' with the confirmation of m to the sender S_j and the destinations. If S_i has no data and sends back a control message, $O(r^2)$ messages are transmitted. Here, S_i sends the control messages to only processes which S_i has not sent the data messages for some predetermined time units without sending the control ones as soon as accepting m to decrease the number of messages.

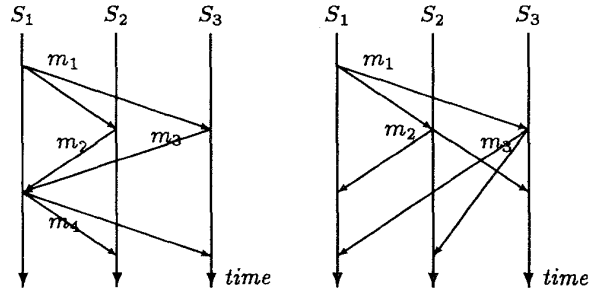

Figure 4: Reliable acceptance

Table 1 shows how to realize the acceptance levels in CBCAST of ISIS [3], CO [18], and SCO protocols. In CBCAST, the network layer supports the continuous acceptance because the network is assumed to be reliable. On the other hand, messages may be lost in the CO and SCO protocols. In order to detect messages lost, the sequence numbers of messages are used.

ISIS uses the vector clock and CO uses the sequence numbers to causally order messages.

3 Data Transmission Procedure

We present the data transmission procedure of the SCO protocol for a group $\mathcal{G} = \langle S_1, ..., S_n \rangle$ by using the high-speed one-to-one network.

3.1 Transmission

Messages are sent to all the processes in $\mathcal{G}$ and only the destinations accept the messages in the SOP [16] and STO [20] protocols since they are sent by using the broadcast network. On the other hand, messages are sent to only the destinations in $\mathcal{G}$ in the SCO protocol since the one-to-one network is used.

S_i sends a message m with *total* sequence number *sqn* and *local* sequence numbers $lsn_1, \ldots, lsn_n$. Each time S_i sends a message m, *sqn* is incremented by one. Here, if m is destined to S_j, lsn_j is incremented by one $(j = 1, \ldots, n)$. m has a field *dst* which denotes the destination processes in $\mathcal{G}$.

S_i manipulates variables SQN, $LSN_1, \ldots, LSN_n$. SQN denotes *sqn* of the message which S_i expects to send next. LSN_j shows lsn_j of the message which S_i expects to send next to S_j.

3.2 Continuous acceptance

S_i manipulates a variable LRN_j which denotes lsn_j of the message which S_i expects to accept next from S_j $(j = 1, ..., n)$. Suppose that S_j sends a message m to S_i. On receipt of m, S_i simply accepts m if $S_i \in m.dst$ and enqueues m into a receipt queue RRQ_j for S_j. Messages from S_j are stored in RRQ_j in the sending order. S_i continuously accepts m if $m.lsn_i = LRN_j$. If $m.lsn_i \neq LRN_j$, S_i finds that S_i does not accept a message m' from S_j where $m.lsn_i > m'.lsn_i \geq LRN_j$. S_i requires S_j to send m' again. Then, S_j sends m' to S_i. On receipt of m', S_i stores m' in RRQ_j so that the messages in RRQ_j are ordered in lsn_i.

Suppose that S_i continuously accepts m from S_j. S_i sends the acceptance confirmation of m to S_j. The acceptance confirmation of m is carried back by messages which S_i sends to S_j. LRN_h, i.e. *sqn* of the message which S_j expects to simply accept next from S_h is stored in $m.ack_h$ $(h = 1, \ldots, n)$. On receipt of m from S_j, S_i knows that S_j has continuously accepted messages from S_h whose $sqn < m.ack_h$.

S_i manipulates an $n \times n$ matrix AL. $m.ack_h$ is stored in AL_{jh} $(h = 1, \ldots, n)$ and $m.sqn$ is in AL_{ij} if m from S_j is accepted by S_i. AL_{jh} denotes *sqn* of the message from S_h which S_i knows S_j expects to continuously accept next.

In the SCO protocol, a message m sent by S_j is sent to only the destinations in $\mathcal{G}$. If S_j sends no message to S_i, S_i cannot know which messages S_j has accepted. S_i sends at least one message to every process every predetermined time units, i.e. control message, if S_i has no data to S_j. S_i has variables $ACC_1, ..., ACC_n$ to denote to which process S_i has to send the acceptance confirmation. Here, if $ACC_j = on$, S_i has not yet sent S_j the acceptance confirmation of the message which S_i had accepted from S_j. If S_i sends some message to S_h, $ACC_h := off$. If ACC_h is still *on* after predetermined time units, S_i sends S_h a control message with $ack_1, ..., ack_n$, and then $ACC_h := on$ for $h = 1, ..., n$. If S_i itself is the destination of m, m is enqueued into RRQ_i and is not sent to S_i.

S_i accepts and sends a message m by the following procedures.

[Acceptance] On receipt of m from S_j,
  if $m.lsn_i = LRN_j$, {
    $AL_{jh} := m.ack_h$ $(h = 1, \ldots, n)$;
    $LRN_j := LRN_j + 1$;
    for $h = 1, \ldots, n$, $ACC_h := on$ if $S_h \in m.dst$;
    m is enqueued into RRQ_j; }
  else *retransmission*; □

[Transmission]
  $m.dst :=$ destinations of m; $m.src := S_i$;
  $m.ack_j := LRN_j$ $(j = 1, ..., n)$;
  $m.sqn := SQN$; $SQN := SQN + 1$;
  for $(j = 1, \ldots, n)$ {
    $m.lsn_j := LSN_j$;
    if $S_j \in m.dst$, $LSN_j := LSN_j + 1$; }
  send m to the destinations; □

3.3 Causal acceptance

Next, we would like to consider how S_i can causally accept the messages. Let m and m' be messages sent to S_i from S_j and S_h, respectively. If every message is sent to all the processes in $\mathcal{G}$, more exactly speaking, if $m'.src \in m.dst$, the following condition [18] holds.

[Causality condition] If $m'.src \in m.dst$, $m \prec m'$ iff
(1) if $S_j = S_h$, $m.sqn < m'.sqn$,
(2) otherwise, $m.sqn < m'.ack_j$. □

In Figure 2, $m_2.sqn \not< m_4.ack_1$ since $m_1.sqn < m_2.sqn$ and $m_4.ack_1 = m_1.sqn + 1$. Thus, the causality condition does not hold unless $P_3 \in m_2.dst$. To order messages in $\prec$, each message m sent by S_j carries the *causal sequence numbers* $csn_1, \ldots, csn_n$, and S_i has variables $CSN_1, \ldots, CSN_n$. On continuous acceptance of m from S_j, S_i manipulates $CSN_1, \ldots, CSN_n$ as follows.

[Causality rule]
(1) $CSN_j := m.sqn + 1$ if $S_j = m.src$.
(2) $CSN_h := max(CSN_h, m.csn_h)$ (for $h = 1, ..., n$, $h \neq j$). □

CSN_j denotes *sqn* of the message from S_j which S_i expects to receive next. Here, $CSN_h \geq AL_{jh}$ $(h = 1, ..., n)$. Thus, the causality number is derived from the total sequence numbers while the vector clock [13] is derived from the local clock. When S_i sends a message m, $m.csn_h := CSN_h$ $(h = 1, \ldots, n)$ in the transmission procedure.

The messages accepted can be causally ordered in the receipt queue by the following theorem.

[Theorem 2] For every pair of messages m and m', $m \prec m'$ iff
(1) $m.sqn < m'.sqn$ if $m.src = m'.src$,

Table 1: Acceptance levels

acceptance level	ISIS (CBCAST)	CO (broadcast)	SCO (point-to-point)
simple	network service	network service	network service
continuous	network service	sequence number	sequence number
causal	vector clock	sequence number	vector of seq. numbers
reliable	decentralized	distributed	distributed

(2) $m.sqn < m'.csn_j$ for $S_j = m.src$ otherwise.

[Proof] If m and m' are sent by the same process, it is clear. Suppose that m is sent by S_j and m' is sent by S_k. First, suppose that $m \prec m'$. From the causality rule, if S_k sends m'' after receiving m, $m.sqn < m''.csn_j$. Thus, $m''.csn_j \leq m'.csn_j$. Hence, $m.sqn < m'.csn_j$.

Next, suppose that $m.sqn < m'.csn_j$. By the causality rule, there is a message m'' such that $m \prec m''$ and $m'' \prec m'$. Hence, $m \prec m'$. □

If S_i simply accepts a message m' from S_j where $m'.lsn_i > LRN_j$, S_i finds that S_i has not continuously accepted a message m where $LRN_j \leq m.lsn_i < m'.lsn_i$. m' is enqueued into RRQ_j and m is transmitted again. The messages in RRQ_j are ordered in $< m_1, ..., m_l >$ where $m_h.sqn_i < m_k.sqn_i$ for $h < k$. Here, let $PRRQ_j$ be a maximally continuous prefix $\langle m_1, ..., m_h \rangle$ of RRQ_j $(h \leq l)$ where m_k is continuously accepted for every $k \leq h$ and m_{h+1} is not if $h < l$. That, S_i does not accept messages sent after m_h before m_{h+1} by S_j

S_i moves messages continuously accepted in RRQ_1, ..., RRQ_n to one *causality* queue CRQ by the following procedure. In CRQ, messages are causally ordered according to Theorem 2.

[Causally ordering procedure]
while ($PRRQ_j \neq \phi$ for j=1, ..., n) {

(1) S_i finds the top message m of some $PRRQ_j$ where $m \prec m'$ for the top m' of every other $PRRQ_h$.

(2) If m is found, m is dequeued from RRQ_j and enqueued into CRQ. If there is the top m' of some $PRRQ_h$ such that $m \parallel m'$, m' is also moved into CRQ. } □

If some $PRRQ_h$ is empty, i.e. no message continuously accepted in RRQ_h, S_i moves no message to CRQ. S_i waits to causally accept m from S_j until S_i continuously accepts a message m' such that $m \preceq m'$ from every process.

[Example] Let us consider a group $\mathcal{G} = \langle S_1, S_2, S_3, S_4 \rangle$ as shown in Figure 6. Here, $s_k\langle s_1, s_2, s_3, s_4 \rangle$ shows a message where $sqn = k$ and $csn_i = s_i$ $(i$=1, ..., 4). Suppose that initially $SQN = 1$ in every process. S_4 continuously accepts messages c_1 from S_3, a_1 from S_1, and c_2 from S_3. S_4 sends d_1 to S_3 and S_4. At (1) of Figure 6, S_4 has four receipt queues $RRQ_1 = \langle a_1 \rangle$, $RRQ_2 = \langle \rangle$, $RRQ_3 = \langle c_1\ c_2 \rangle$, and $RRQ_4 = \langle d_1 \rangle$. Since $PRRQ_2$ is empty, no message is removed from any receipt queue. Then, S_4 continuously accepts a_2 from S_1, and b_2 from S_2. At (2), $RRQ_1 = \langle a_1\ a_2 \rangle$, $RRQ_2 = \langle b_2 \rangle$, $RRQ_3 = \langle c_1\ c_2 \rangle$, and $RRQ_4 = \langle d_1 \rangle$. The top messages a_1, b_2, c_1, and d_1 in the receipt queues are compared on csn. Here, $PRRQ_i = RRQ_i$ since there is no message loss (i = 1, ..., 4). a_1 and c_1 are removed from RRQ_1 and RRQ_3, respectively, and are enqueued into CRQ since $a_1 \parallel c_1$, $a_1 \prec b_2$, and $a_1 \prec d_1$. Here, $RRQ_1 = \langle a_2 \rangle$, $RRQ_2 = \langle b_2 \rangle$, $RRQ_3 = \langle c_2 \rangle$, and $RRQ_4 = \langle d_1 \rangle$. The tops a_2, b_2, c_2, and d_1 in the receipt queues are compared again. Since $c_2 \parallel d_1$, $c_2 \prec a_2$, and $c_2 \prec b_2$, c_2 and d_1 are moved into the causality queue CRQ. Here, $CRQ = \langle c_1\ a_1\ c_2\ d_1 \rangle$ where $c_1 \preceq a_1 \prec c_2 \preceq d_1$. Here, CRQ might be $\langle a_1\ c_1\ c_2\ d_1 \rangle$, $\langle a_1\ c_1\ d_1\ c_2 \rangle$, or $\langle c_1\ a_1\ d_1\ c_2 \rangle$ since $a_1 \parallel c_1$ and $c_2 \parallel d_1$.

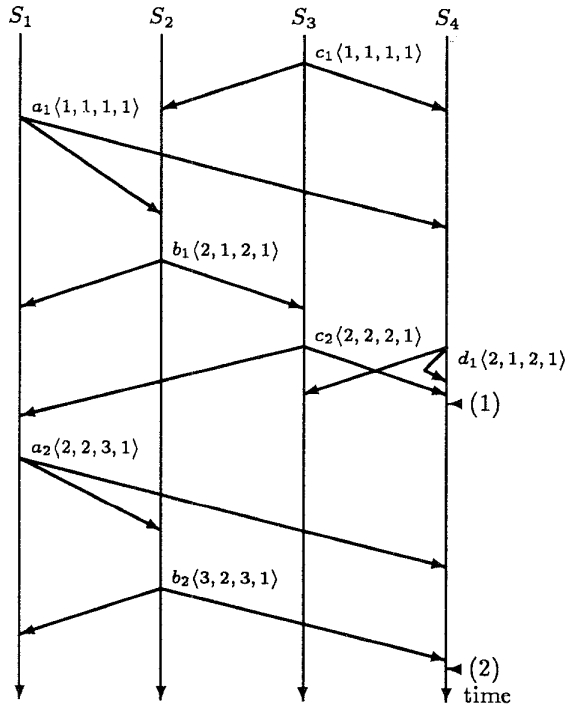

Figure 6: Example

Here, suppose that S_4 loses a_1. S_4 finds the loss of a_1 on acceptance of a_2. $PRRQ_1$ is empty while $RRQ_1 = \langle a_2 \rangle$. Hence, no message in RRQs is moved to CRQ. On receipt of a_1, S_4 obtains the same receipt queues as (2). Then, the messages are causally

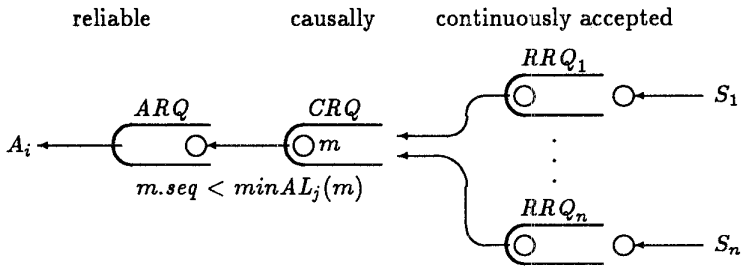

Figure 5: Queues in S_i

accepted. □

[Proposition 3] For every message m in CRQ, S_i causally accepts m.

[Proof] It is clear from Theorem 1. □

Each time a message m from S_j is moved to CRQ, $AL_{ij} := m.sqn$. All the messages from every S_j whose $sqn < AL_{ij}$ are causally accepted by S_i.

3.4 Full acceptance

S_i does not yet know whether messages in CRQ are reliably accepted by the destinations. Let m be a message accepted by S_i from S_j and $minAL_j(m)$ be $min(\{AL_{hj} \mid S_h \in m.dst\})$. That is, S_i knows that every destination of m has continuously accepted messages from S_j whose $sqn < minAL_j(m)$. Hence, if $m.sqn < minAL_j(m)$, m is causally reliably accepted. m is fully accepted since m can be delivered to the application.

[Theorem 4] If S_i reliably accepts m, m is eventually reliably accepted by every destination of m.

[Proof] Suppose that S_i reliably accepts m but S_j does not. S_j loses a message m' confirming m from some destination S_h of m. By the time out mechanism of the acceptance procedure, S_h retransmits m' to S_h. Then, S_j reliably accepts m. □

[Theorem 5] For every message m in ARQ, S_i fully accepts m.

[Proof] From Proposition 1 and Theorem 4, the messages in ARQ are causally reliably accepted. □

S_i has one *acknowledgment* queue ARQ in which messages causally reliably accepted, i.e. fully accepted are stored. While the top m of CRQ, where $m.dst = S_j$, satisfies $m.sqn < minAL_j(m)$, m is dequeued from CRQ and enqueued into ARQ.

[Full acceptance]

while ($m.sqn < minAL_j(m)$ for the top m of CRQ, where m is sent by S_j) {
  m is dequeued from CRQ and
  enqueued into ARQ. } □

The messages are causally reliably delivered to the application process A_i by dequeuing the messages from ARQ.

3.5 Flow control

Each S_i has a finite number of buffers to accept messages. S_i has a variable BUF_j which denotes the number of available buffers in S_j ($j = 1, ..., n$). S_i includes the number of available buffers in the field *buf* of m, i.e. $m.buf := BUF_i$. On acceptance of m' from S_j, S_i knows how many available buffers S_j has and $BUF_j := m'.buf$. If more than BUF_i messages arrive at S_i, S_i loses messages due to the buffer overflow. In the SCO protocol, messages are sent to only the destinations. Let $minBF(m)$ denote $min\{ BUF_h \mid S_h \in m.dst \}$. Here, suppose that $BUF_h = minBF(m)$. If every process sends BUF_h messages to S_h, the buffer overflow occurs in S_h. Here, if S_i sends one nth of BUF_h messages, S_h can receive the messages without the buffer overflows even if others send messages to S_h. Therefore, each S_i can send a message m only while the following flow condition is satisfied. Here, W is the maximum window size and H is constant (> 1).

[Flow condition] $minAL_i \leq SQN < minAL_i + min(W, minBF(m) / (H \times n))$. □

4 Evaluation

We assume that each process sends messages randomly to r ($\leq n$) processes in a group $\mathcal{G} = \langle S_1, ..., S_n \rangle$. In the non-distributed protocol, i.e. centralized or decentralized protocol, one coordinator sends a message m to the destinations in $\mathcal{G}$ and the destinations send back the confirmation messages to the coordinator if they succeed in simply accepting m. The coordinator sends the acceptance confirmation of m if all the destinations receive m, otherwise sends the failure to them. Hence, simply accepts $3r$ messages are transmitted and it takes three rounds for each message to be reliably accepted as shown in Figure 4(1).

In the distributed protocol, after accepting a message m from S_j, each destination S_i of m sends the acceptance confirmation to S_j and all the destinations as shown in Figure 4(2). r^2 messages are transmitted and it takes two rounds for each message to be reliably accepted. In order to reduce the number of messages, the confirmation of m is *carried back* by the messages sent by S_i. S_i does not send the confirmation of m as soon as simply accepting m, i.e. *deferred confirmation*. S_i sends the confirmation to S_k if S_i does not send messages to S_k for predetermined time units. The same modification can be adopted to the centralized protocol. The distributed and non-distributed protocols with piggy back and deferred confirmation are named *modified* distributed and non-distributed

ones, respectively.

Four control schemes, non-distributed (C_0), modified non-distributed (C_1), distributed (D_0), and modified distributed (D_1) ones are compared in terms of number of messages and delay to fully accept a message. D_1 means the SCO protocol. Figures 7 and 8 show the ratios of the number of messages and the delay of C_1, D_0, and D_1 to C_0 where n=10. Here, we assume that every process S_i sends a ($\geq$ 1) messages every one time unit. We also assume that S_i sends the control messages to only the processes to which S_i does not send messages in k ($\geq$ 1) time units. One round is t time units. Figures 7 and 8 show the ratios. for k, where a=1, t=4, and r=5, i.e. each process sends *one* (a) data message every time unit and sends control messages if there are processes to which S_i has sent no data message for k time units, and it takes *four* (t) time units to deliver a message to the destination. Figure 7 shows that the longer k is, the lower number of control messages are transmitted but the longer delay it takes. However, D_1 does not require much longer delay than D_0 and the delay of D_1 is much smaller than C_0 and C_1. C_1 has the minimum number of messages but the longest response time, about two times longer than D_1.

Figures 9 and 10 show the ratios of the number of messages and the delay for the number r of the destinations where $a = 1$, $k = 4$, and $t = 4$. D_1 has lower number of messages than C_0 and D_0. For r = 3 to 8, C_1 has lower number of messages than D_1 but the difference between C_1 and D_1 is smaller than 20% of D_1. The delay of D_1 is 50% smaller than C_0 and C_1 and does not get much greater than D_0 while the deferred confirmation is used in D_1.

In summary, C_1 has the lowest number of messages but the longest delay. D_0 has the shortest delay but the largest number of messages. D_1, i.e. SCO supports the second lowest number of messages and the second shortest delay, but the difference from the best one is small, i.e. smaller than 10%. Hence, D_1 can support the better feature than the others in terms of number of messages and delay.

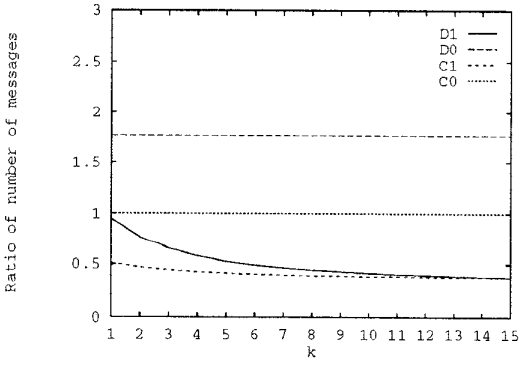

Figure 7: Number of messages ($n = 10$)

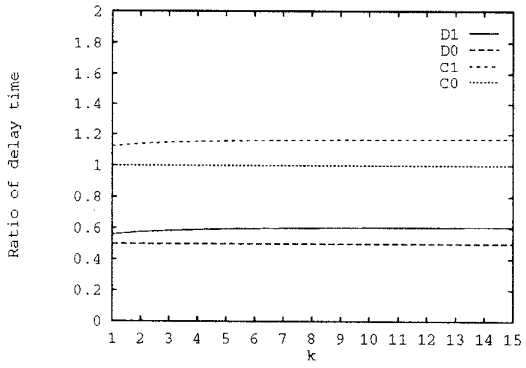

Figure 8: Delay ($n = 10$)

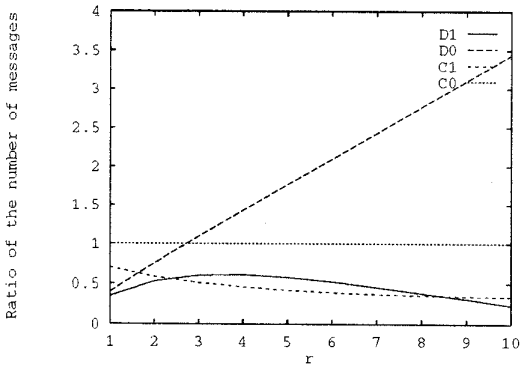

Figure 9: Number of messages ($n = 10$)

5 Concluding Remarks

This paper has discussed the intra-group communication (SCO) protocol which supports the causally ordered and selective delivery of messages to the destinations in the group. The SCO protocol uses the high-speed one-to-one network where the system processes may not receive messages due to buffer overrun and congestion. Messages are sent to only destinations in the group while they are sent to all the processes in [20]. The SCO protocol is based on the distributed control, where each process sends messages to the sender and destinations carrying the acceptance confirmation of the messages which the process has accepted. In order to reduce the number of messages, the SCO protocol adopts the deferred confirmation and the piggy back. We have shown that lower number of messages are transmitted and it takes shorter delay in the SCO protocol than the non-distributed ones.

References

[1] Abeysundara, B. W. and Kamal, A. E., "High-Speed Local Area Networks and Their Performance: A Survey," *ACM Computing Surveys*, Vol.23, No.2, 1991, pp.221-264.

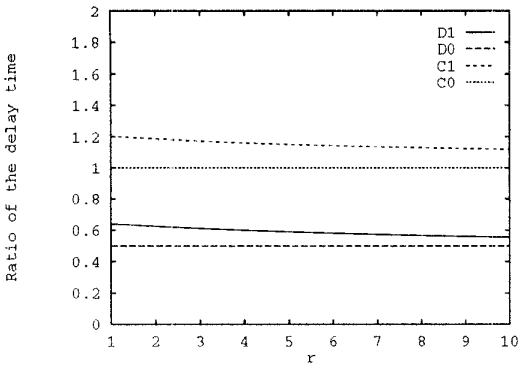

Figure 10: Delay ($n = 10$)

[2] 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.

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

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

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

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

[7] Gray, J., "Notes on Database Operating Systems, An Advanced Course," *Lecture Notes in Computer Science, Springer-Verlag*, No.60, 1978, pp.393-481.

[8] Hadzilacos, V. and Toueg, S., "Fault-Tolerant Broadcasts and Related Problems," *Distributed Systems (2nd ed.), Mullender, S. ed., Addison-Wesley*, 1993, pp.97-145.

[9] Kaashoek, M. F. and Tanenbaum, A. S., "Group Communication in the Amoeba Distributed Operating System," *Proc. of the 11th IEEE ICDCS*, 1991, pp.222-230.

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

[11] Leong, H.V., and Agarwal, D., "Using Message Semantics Reduce Rollback in Optimistic Message Logging Recovery Schemes," *Proc. of the 14th IEEE ICDCS*, 1994, pp.227-234.

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

[13] 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.

[14] 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.

[15] Moser, L.E., Amir, Y., Melliar-Smith, P.M., and Agarwwal, D.A., "Extended Virtual Synchrony," *Proc. of the 14th IEEE ICDCS*, 1994, pp.56-65.

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

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

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

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

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

[21] Takizawa, M., "Design of Highly Reliable Broadcast Communication Protocol," *Proc. of the IEEE COMPSAC'87*, 1987, pp.731-740.

[22] Takizawa, M. and Nakamura, A., "Partially Ordering Broadcast (PO) Protocol," *Proc. of the 9th IEEE INFOCOM*, 1990, pp.357-364.

[23] Takizawa, M. and Nakamura, A., "Reliable Broadcast Communication," *Proc. of IPSJ Int'l Conf. on Information Technology (InfoJapan)*, 1990, pp.325-332.

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