ソフトウェア基礎論　37－10
プログラミング言語　27－10
（1990. 12. 13）

# 非同期分散システムにおける瞬時メッセージ転送

曽根岡　昭直
NTT ソフトウェア研究所

**あらまし**　分散システムのプロトコル設計には非同期性（プロセス間通信遅延時間が有界でないこと）に起因する難しさがある。アプリケーションプロトコルの設計を容易化するために、本稿では瞬時メッセージ転送を模擬する転送方式を提案する。この転送方式は、以下の特徴を持つ。1) 各プロセスがクライアントともサーバとも動作するモデルにおいても、デッドロック状態に陥ることなく適用可能。2) 受信プロセスとの同意をとることなく送信が可能。3) デッドロックの可能性がない場合にはユーザメッセージ当たり 1 つの ACK だけで、デッドロックが（$k$ 個のユーザメッセージから）生じた場合にも最悪 $k(k+1)/2$ 個のシステムメッセージの転送で済む。

# Instantaneous Message Passing in Asynchronous Distributed Systems

**Terunao SONEOKA**
NTT Software Laboratories
Midori-cho 3-9-11, Musashino-shi, Tokyo 180, Japan

**Abstract**　Asynchrony (unbounded message transmission delay) complicates the design of protocols for distributed systems. For simplifying the design task, this paper proposes an interprocess communication mechanism for *simulating* instantaneous message passing. This mechanism has the following properties. 1) It is applicable without deadlock for the model where each process acts as both client and server. 2) It allows each sender to send a message without coordinating with the receiver. 3) It requires only an ACK for each user message if there is no possibility of deadlock; even if deadlock occurs, at worst $k(k+1)/2$ system messages are transmitted for $k$ user messages in a deadlock cycle.

## 1 Introduction

Distributed systems with no known bound on relative process speeds or message transmission delays are called *asynchronous*. Asynchrony makes coordination between processes difficult, and complicates the design and verification of protocols for such systems. These difficulties can be reduced if one can assume that every message passing is achieved instantaneously.

Typical coordination errors in asynchronous systems are illustrated as follows.

- An executive manager $A$ first sends engineer $C$ a message $m_1$ "Please do a job $X$", and in an hour sends message $m_2$ ordering a division head $B$ to check $C$'s progress on the job $X$. On receiving $m_2$, $B$ asks $C$'s progress by sending message $m_3$. However, when $C$ receives $m_3$, $C$ has not yet received $m_1$ and is confused about what $B$ asks. This situation corresponds to the violation of *causal ordering* in [2, 9] (see Fig. 1 (a)).
- $A$ sends his girl friend $B$ a message $m_1$ "I will pick you up at your office at 5:00 PM". Concurrently, $B$ sends $A$ a message $m_2$ "We will meet at the theater at 5:00 PM". Thus, $A$ goes directly to the theater, but $B$ keeps waiting for $A$ at her office. This situation corresponds to the *process collision* in [6] (see Fig. 1(b)).
- An executive manger $A$ first sends division head $B$ a message $m_1$ "Sorry, I have changed my mind. We will have a meeting at 1:00 PM". Concurrently, $B$ send engineer $C$ a message $m_3$ "Today's meeting is put off". After sending $m_1$, $A$ sends $C$ a message $m_2$ "I would like to hear your opinion at today's meeting". However, $C$ receives $m_2$ before receiving $m_3$. Thus, $C$ misunderstands that the meeting is put off (see Fig. 1(c)).

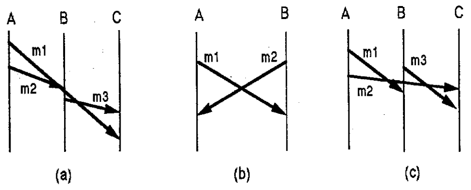

Figure 1: Examples of coordination errors in asynchronous distributed systems

All of these errors can be easily removed if every message passing is achieved instantaneously. Unfortunately, one cannot implement a system with instantaneous message passing. Along the lines of the abstraction techniques simplifying the design task in [8], this paper proposes a mechanism for *simulating* instantaneous message passing in distributed systems. Here, simulating means that each process in a system *cannot distinguish* whether or not the system is using an ideal instantaneous message passing.

This paper first defines the concept of "*indistinguishable*". This concept gives a taxonomy of synchronous message passing mechanisms and clarifies that the proposed mechanism - which is indistinguishable from instantaneous message passing - achieves higher synchrony than the causal ordering in [2, 9]. We then proposes an algorithm for it. This algorithm is deadlock-free and attains fairness. It also allows an $n$-process system to have $n-1$ concurrent messages. It requires only an acknowledgement message for each user message if there is no possibility of deadlock. When deadlock occurs with $k$ user messages in a deadlock cycle, at worst $k(k+1)/2$ system messages are transmitted; the order of this number is equal to the best known message complexity of deadlock detection algorithm[10].

The related communication primitives are the *remote procedure call* (RPC) and the synchronous message passing for the generalized alternative command suggested in CSP[3] (referred as *rendezvous* in [1]). Although the RPC is widely used for the *client/server model*, it might bring about deadlocks in the *partner model*, where each process acts as both client and server. The proposed instantaneous message passing mechanism is applicable without deadlock even for the partner model. In the rendezvous, on the other hand, before executing a message passing, the sender and receiver must both be ready. Thus, the concurrency of the algorithms for rendezvous(cf. [1]) is low in the sense that at most $\frac{n}{2}$ processes can send messages concurrently in an $n$-process system.

The rest of the paper is organized as follows. Section 2 presents the model of distributed systems and the related definitions. Section 3 proposes a message passing mechanism indistinguishable from instantaneous message passing and also clarifies its position in a synchronous message passing taxonomy. Section 4 proposes an algorithm for this mechanism and proves its correctness, fairness, and freedom from deadlock. This section also evaluates the complexity of this algorithm. Section 5 shows an application to the resolution of process collision in communication protocols and gives a preliminary analysis of the run-time overhead.

## 2 Model and definitions

### 2.1 Model

We consider an *asynchronous* distributed system of $n$ processes that communicate through message passing. For the sake of simplicity, we assume that processes are fully connected. By asynchronous, we mean that there is (1) no global clock, (2) no bound on message transmission delay, and (3) no assumption about the relative speed of processes. We assume that (1) processes and channels are reliable (every message transmitted is eventually received), (2) channels are FIFO, and (3) each process has a distinct ID. Let $P = \{1, \ldots, n\}$ be the set of process ID's in the system.

The system consists of two parts: an *application layer* with user processes, and a *message-passing layer* with control processes and channels. For sending a message $m$ to another user process $q$, user process $p$ issues $send_p^q(m)$ (or, shortly, $send_p(m)$) to its control process $c_p$. Upon receiving it, $c_p$ transmits $m$ to another control process $c_q$ (denoted by $trans_p^q(m)$ or, shortly, $trans_p(m)$) if some condition is satisfied. On receiving the message $m$ (denoted by $recv_q^p(m)$ or, shortly, $recv_q(m)$), $c_q$ delivers $m$ to its user process $q$ (denoted by $delv_q^p(m)$ or, shortly, $delv_q(m)$) if some condition is satisfied.

Besides *send event* and *deliver event*, user processes execute *internal event*. Send event and internal event are the results of actions autonomously taken by the process. Deliver event is not explicitly controlled by the process. Let us assume that each event takes exactly one local time unit to execute.

### 2.2 History

To define "simulate", we prepare the following concepts. A specific execution of a system is described by a *history*. A history $H$ consists of the following four history functions; $H = < C, Q, A, B >$.

- The *clock history function* $C$ maps from processes $P$ and real times $R$ to clock times $N$; i.e., $C : P \times R \mapsto N$. $C(p,t)$ is the time on $p$'s clock at real time $t$. Since a process's clock never decreases, clock history functions satisfy the following condition:

$$CC : \forall p \in P \forall t_1, t_2 \in R\ [t_1 < t_2 \Rightarrow C(p,t_1) \le C(p,t_2)].$$

- The *state history function* $Q$ maps from processes and clock times to process states $S$; i.e., $Q : P \times N \mapsto S$. Process $p$ is in state $Q(p,c)$ when its clock shows $c$.
- The *event history function* $A$ maps from processes and clock times to events $E$; i.e., $A : P \times N \mapsto E$. This is a partial function, since processes do not have events at every instant of clock time. Process $p$ has event $A(p,c)$ when its clock shows $c$.
- The *channel history function* $B$ maps from pairs of processes and real times to message sequences; i.e., $B : P \times P \times R \mapsto M^s$, where $M$ is the message set and $s$ is the channel size. For example, if $B(p,q,t) = \{m_1, m_2\}$, then the channel from $q$ to $p$ contains messages $m_1$ and $m_2$.

Two histories $H_1 =< C_1, Q_1, A_1, B_1 >$ and $H_2 =< C_2, Q_2, A_2, B_2 >$ are *equivalent*, denoted by $H_1 \sim H_2$, if $Q_1 = Q_2$ and $A_1 = A_2$; otherwise, we denote $H_1 \not\sim H_2$. Informally, $H_1 \sim H_2$ if in both histories each process executes the same events from the same states at the same local times. Since processes can observe neither real time (they can only observe their local clocks) nor the contents of channels, they cannot distinguish $H_1$ from $H_2$.

A distributed system is identified by the set of histories that correspond to all executions of the system. Each process $p$ runs a *local protocol*, $(\delta_p, \Pi_p)$. $\delta_p$ determines the next state of the process based on its present state and the event executed; i.e., $\delta_p : S \times E \mapsto S$. If $p$ is in state $s$ and event $a$ occurs, then it changes to state $\delta_p(s,a)$. Given $p$'s clock and state, $\Pi_p$ specifies the next event candidates executed by $p$; i.e., $\Pi_p : N \times S \mapsto 2^E$. If $p$ is in state $s$ at local time $c$, then $\Pi_p$ specifies that it executes an event in $\Pi_p(c,s)$. A set of local protocols, $\Pi = \{(\delta_p, \Pi_p) | p \in P\}$, is a *protocol*.

History $H =< C, Q, A, B >$ is *consistent with* protocol $\Pi$ if

$$\begin{aligned}\forall p \in P \forall c \in N\ \ [\ \ & A(p,c) \text{ is defined} \Rightarrow \\ & Q(p, c+1) = \delta_p(Q(p,c), A(p,c)) \\ & \wedge A(p,c) \in \Pi_p(c, Q(p,c))]\end{aligned}$$

This definition leads to the following[8]:

**Lemma 2.1** *If $H_1 \sim H_2$ and $H_1$ is consistent with protocol $\Pi$, then $H_2$ is also.*

A distributed system problem is specified by a predicate $\Sigma$ on histories. This predicate is the problem's *specification*. Protocol $\Pi$ solves a problem with specification $\Sigma$ in system $S$ if whenever processes run $\Pi$ in $S$, the resulting history satisfies $\Sigma$.

Many problems in distributed systems have specifications that make no reference to real time. For example, one can specify that transaction execution in a distributed database is serializable without referring to real time. This notion is formalized with *internal specifications.* A specification $\Sigma$ is *internal* if for any two equivalent histories $H_1 \sim H_2$, $H_1$ satisfies $\Sigma$ if and only if $H_2$ does. A specification that does not refer to real time or to the channel contents must be internal.

### 2.3 Relations between events and messages

The *"happens-before"* relation "$\rightarrow$" is defined on the set of events of a system by Lamport[5] as follows.

**Definition 2.1** *Event $e_1$* **happens-before** *event $e_2$, denoted by $e_1 \rightarrow e_2$, iff one of the following conditions is true.*

1. *$e_1$ and $e_2$ occur on the same process $p$, and $e_1$ precedes $e_2$ in its local time (denoted by $e_1 \xrightarrow{p} e_2$).*
2. *$e_1$ is the sending of a message and $e_2$ is the delivery of that message.*
3. *transitive closure of 1 and 2.*

Note that $e_1 \not\rightarrow e_1$ for any event $e_1$ (*irreflexiveness*).

Further, we define the *"exists-before"* relation "$\succ$" on the set of messages in a system as follows.

**Definition 2.2** *Message $m_1$* **exists-before** *message $m_2$, denoted by $m_1 \succ m_2$, iff one of the following conditions is true.*

1. *$m_1$ and $m_2$ are sent by the same process $p$ and $send_p(m_1) \xrightarrow{p} send_p(m_2)$ (denoted by $-m_1 \succ_p -m_2$ and called $(-,-)$ relation).*
2. *$m_1$ and $m_2$ are respectively sent and delivered by the same process $p$ and $send_p(m_1) \xrightarrow{p} delv_p(m_2)$ (denoted by $-m_1 \succ_p +m_2$ and called $(-,+)$ relation).*
3. *$m_1$ and $m_2$ are respectively delivered and sent by the same process $p$ and $delv_p(m_1) \xrightarrow{p} send_p(m_2)$ (denoted by $+m_1 \succ_p -m_2$ and called $(+,-)$ relation).*
4. *$m_1$ and $m_2$ are delivered by the same process $p$ and $delv_p(m_1) \xrightarrow{p} delv_p(m_2)$ (denoted by $+m_1 \succ_p +m_2$ and called $(+,+)$ relation).*
5. *transitive closure of 1, 2, 3, and 4.*

Especially, we denote $m_1 \succ_p m_2$ iff one of the above conditions 1, 2, 3, or 4 holds.

## 3 Taxonomy of synchronous message passing

This section proposes several categories of synchronous message passing. We consider systems with different message passing: $S(M)$ denotes a system using a message passing $M$. To clarify the relationship between a system $S(MP_1)$ using a message passing $MP_1$ and a system $S(MP_2)$ using a distinct message passing $MP_2$, we define the following terminology: $S(MP_1)$ is *indistinguishable from $S(MP_2)$*, denoted by $S(MP_1) \trianglelefteq S(MP_2)$, (or simply, $MP_1$ is indistinguishable from $MP_2$) if

$$\forall H \in S(MP_1) \exists H' \in S(MP_2)[H \sim H'].$$

Clearly, $S(MP_1) \trianglelefteq S(MP_2)$ if $S(MP_1) \subseteq S(MP_2)$; that is, $H \in S(MP_1) \Rightarrow H \in S(MP_2)$. It is said that $S(MP_1)$ is *distinguishable from $S(MP_2)$*, denoted by $S(MP_1) \not\trianglelefteq S(MP_2)$, if $\exists H \in S(MP_1) \forall H' \in S(MP_2)[H \not\sim H']$.

We will consider the following ideal synchronous message passing categories $I_1$, $I_2$, and $I_3$.

$I_1$: **$\Delta_{p,q}$-consistency.** For any channel from process $p$ to process $q$, the message transmission delay of that channel (i.e., the time interval between $send_p(m)$ and $delv_q(m)$) is always equal to the same value $\Delta_{p,q}$ $(\ge 0)$.

$I_2$: **$\Delta_p$-consistency.** For any process $p$, the message transmission delay from $p$ to any other process $q$ is always equal to the same value $\Delta_p$ $(\ge 0)$.

$I_3$: **Instantaneous Message Passing.** For any message $m$, its transmission delay is always equal to zero; that is, every message is transmitted instantaneously.

Also, the following actual message-passing methods $MP_a$, $MP_b$, and $MP_c$ are considered. The algorithms for causal delivery ($MP_b$) are proposed in [2, 9]. An algorithm for no message-crossing ($MP_c$) will be proposed in the following section.

$MP_a$: **FIFO.** If $send^q_p(m) \xrightarrow{p} send^q_p(m')$, then $delv^p_q(m) \xrightarrow{q} delv^p_q(m')$.

$MP_b$: **Causal ordering.** If $send^r_p(m) \rightarrow send^r_q(m')$, then $delv^p_r(m) \xrightarrow{r} delv^r_q(m')$.

$MP_c$: **No message-crossing.** There is no pair of messages $m$ and $m'$ such that $(m \succ m') \wedge (m' \succ m)$.

The following relations are satisfied (see Fig. 2).

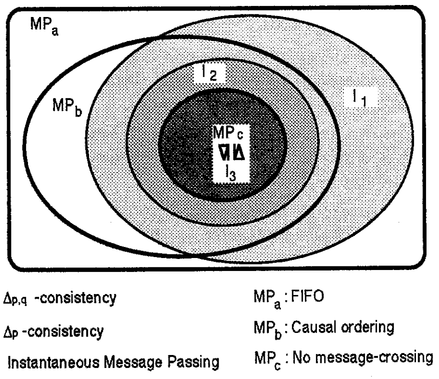
MP$_a$
MP$_b$
I$_2$
I$_1$
MP$_c$
$\trianglerighteq\trianglelefteq$
I$_3$
I$_1$ : $\Delta_{p,q}$ -consistency
I$_2$ : $\Delta_p$ -consistency
I$_3$: Instantaneous Message Passing
MP$_a$ : FIFO
MP$_b$ : Causal ordering
MP$_c$ : No message-crossing

Figure 2: Relations among synchronous message-passing categories

**Theorem 3.1** *1.* $S(I_1) \subseteq S(MP_a)$. $S(MP_a) \not\trianglelefteq S(I_1)$.

*2.* $S(I_2) \subseteq S(MP_b)$. $S(MP_b) \not\trianglelefteq S(I_2)$.

*3.* $S(I_2) \subseteq S(I_1)$. $S(I_1) \not\trianglelefteq S(I_2)$.

*4.* $S(MP_b) \subseteq S(MP_a)$. $S(MP_a) \not\trianglelefteq S(MP_b)$.

*5.* $S(I_1) \not\trianglelefteq S(MP_b)$. $S(MP_b) \not\trianglelefteq S(I_1)$.

*6.* $S(I_3) \subseteq S(I_2)$. $S(I_2) \not\trianglelefteq S(I_3)$.

*7.* $S(I_3) \subseteq S(MP_c)$. $S(MP_c) \trianglelefteq S(I_3)$.

**Proof:**

1. Consider any history $H \in S(I_1)$. Let $m_1$ and $m_2$ be any two messages from process $p$ to process $q$ such that $send_p(m_1) \xrightarrow{p} send_p(m_2)$. Let $A(p,c_1) ="\ send_p(m_1)"$, $A(p,c_2) ="\ send_p(m_2)"$, $A(q,c_3) ="\ delv_q(m_1)"$, $A(q,c_4) ="\ delv_q(m_2)"$, and $t_i$ $(i = 1,\ldots,4)$ be the real time each event occurs such that $C(p,t_i) = c_i$ $(i = 1,2)$ and $C(q,t_i) = c_i$ $(i = 3,4)$. From $c_1 < c_2$ and clock condition $CC$, we get $t_1 < t_2$. Since $H \in S(I_1)$, $t_3 = t_1 + \Delta_{p,q}$ and $t_4 = t_2 + \Delta_{p,q}$. Thus, $t_3 < t_4$. From clock condition $CC$ and $c_3 \neq c_4$, we get $c_3 < c_4$; that is, $delv_q(m_1) \xrightarrow{q} delv_q(m_2)$. Therefore, $H \in S(MP_a)$. On the other hand, let us consider $H_1 \in S(MP_a)$ shown in Fig. 3(a). For $H_1$, there is not an $H_1' \in S(I_1)$ such that $H_1' \sim H_1$ because we get $\Delta_{p,r} > \Delta_{p,q}$ from the relations among messages $m_1$, $m_2$, and $m_3$, and we get the contradiction $\Delta_{p,q} > \Delta_{p,r}$ from the relations among messages $m_4$, $m_5$, and $m_6$.
2. Similarly, it is easy to show that $H \in S(I_2) \Rightarrow H \in S(MP_b)$. On the other hand, let us consider $H_2 \in S(MP_b)$ shown in Fig. 3(b). For $H_2$, there is not an $H_2' \in S(I_2)$ such that $H_2' \sim H_2$ because we get $\Delta_r > \Delta_q$ from the relations among messages $m_1$, $m_2$, $m_3$, and $m_4$, and we get $\Delta_q > \Delta_r$ from the relations among messages $m_5$, $m_6$, $m_7$, and $m_8$.
3, 4. It is clear that $H \in S(I_2) \Rightarrow H \in S(I_1)$ and $H \in S(MP_b) \Rightarrow H \in S(MP_a)$. On the other hand, $H_3 \in S(I_1)$ (or $S(MP_a)$) shown in Fig. 3(c) is a counterexample of $S(I_1) \trianglelefteq S(I_2)$ $(S(MP_a) \trianglelefteq S(MP_b))$.
5. $H_3 \in S(I_1)$ shown in Fig. 3(c) is also a counterexample of $S(I_1) \trianglelefteq S(MP_b)$. On the other hand, $H_2 \in S(MP_b)$ shown in Fig. 3(b) is also a counterexample of $S(MP_b) \trianglelefteq S(I_1)$, because we get $\Delta_{p,r} + \Delta_{r,s} > \Delta_{p,q} + \Delta_{q,s}$ from the relations among messages $m_1$, $m_2$ ,$m_3$, and $m_4$, and we get $\Delta_{p,r} + \Delta_{r,s} < \Delta_{p,q} + \Delta_{q,s}$ from the relations among messages $m_5$, $m_6$, $m_7$, and $m_8$.
6. It is clear that $H \in S(I_3) \Rightarrow H \in S(I_2)$. On the other hand, consider $H_4 \in S(I_2)$ shown in Fig. 3(d). For $H_4$, there is not an $H_4' \in S(I_3)$ such that $H_4' \sim H_4$.
7. Consider any $H \in S(I_3)$. Every message in $H$ can be partially ordered according to the real time instant $t$ when its message passing occurs. Thus, it is clear that there is no pair of messages $m$ and $m'$ in $H$ such that $(m \succ m') \wedge (m' \succ m)$. On the other hand, consider any $H' \in S(MP_c)$, where every message can be partially ordered according to the relation $m \succ m'$. Let us assign any real time $t_i \in R$ to message $m_i$ such that $m_i \succ m_j \Rightarrow t_i < t_j$. From this, it is clear that the clock of any process $p$, $C(p,t)$, satisfies the clock condition $CC$; that is, $t_1 < t_2 \Rightarrow C(p,t_1) \leq C(p,t_2)$. Thus, there exists $H'' \in S(I_3)$ such that $H' \sim H''$.

□

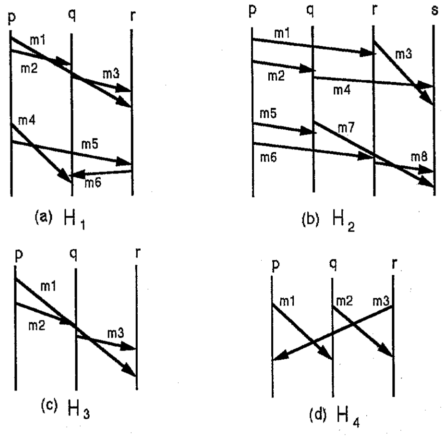

Figure 3: Counterexamples

Suppose that a protocol designer derives and proves a protocol for a system with an ideal synchronous message passing $I$. That is, the designer proves that all histories in $S(I)$ consistent with this protocol satisfy a given specification $\Sigma$. If the protocol is run in a different system that does not have the ideal message passing $I$, then it might no longer satisfy $\Sigma$. The following theorem shows that if $\Sigma$ is internal and if a system uses the message passing $MP$ that is indistinguishable from $I$ (i.e., system $S(MP)$ such that $S(MP) \trianglelefteq S(I)$ is used), then the protocol remains correct.

**Theorem 3.2** *Let $\Sigma$ be an internal specification. Let $\Pi$ be a protocol that satisfies $\Sigma$ when run in a system with an ideal synchronous message passing $I$, $S(I)$. Then $\Pi$ also satisfies $\Sigma$ when run in a system $S(MP)$ that is indistinguishable from $S(I)$.*

**Proof:** By the assumption on $\Pi$, any $H' \in S(I)$ that is consistent with $\Pi$ satisfies $\Sigma$. Consider an $H \in S(MP)$ consistent with $\Pi$. Since $MP$ is indistinguishable from $I$, there is an $H' \in S(I)$ such that $H' \sim H$. By Lemma 2.1, $H'$ is also consistent with $\Pi$. By the assumption, $H'$ satisfies $\Sigma$. Since $\Sigma$ is internal and $H' \sim H$, $H$ also satisfies $\Sigma$.

□

## 4 Algorithm simulating instantaneous message passing

This section shows an algorithm with no message-crossing $S(MP_c)$ for simulating instantaneous message passing $S(I_3)$. Here, the message-passing layer of a system is considered, and control processes are simply called processes. On receiving $send_p^q(m)$ from a user process $u_p$, the control process $p$ sets $willsend_p(q; m) := true$.

### 4.1 Algorithm

As in Fig. 1(a),(c), if a process can send messages $m_1$ and $m_2$ successively, $m_3$ might be received earlier than $m_1$ or later than $m_2$ because of unknown message transmission delays, which brings about message-crossing. Our algorithm uses an acknowledgement message **ACK**, which is similar to the RPC, to control the successive sends. On sending a message $m$ to process $q$, process $p$ changes its state from **NORMAL** to **WAIT**. It thereby becomes unable to send any successive messages until receiving the ACK of $m$, by which $p$ changes its state back to NORMAL. On receiving a message $m$ from $p$, process $q$ delivers it and transmits the ACK to $p$ if its state is NORMAL; otherwise, $q$ waits to transmit the ACK until its changes to NORMAL. When its state changes to NORMAL by receiving the ACK from $q$, process $p$ delivers every message $m'$ received during the WAIT state and transmits the ACK of $m'$ to its sender.

If two processes $p$ and $q$ send messages to each other concurrently, however, they have to wait forever before transmitting the ACK; that is, deadlock occurs. To resolve this, we use a deadlock detection algorithm similar to that proposed by Mitchell and Merritt[7]. The proposed algorithm is modified from theirs to satisfy the following *fairness* condition without using an additional phase.

**Fairness.** If process $p$ continually wishes to send a message to another process $q$, then $p$ will eventually be able to send a message to $q$.

In the deadlock detection algorithm, the system can be described by the *wait-for graph* (WFG), a directed graph in which each node represents a process and a directed edge $(p, q)$ indicates that $q$ blocks $p$ by receiving a message $m$ from $p$ while its state is WAIT. Note that the maximum outdegree of the WFG is one, since each process can send only one message before receiving its ACK. A system is deadlocked when a cycle forms in its WFG. In the proposed algorithm, probes are sent in the direction opposite the edges in the WFG, where a **PROBE** contains a natural number unique to the nodes in the WFG. Simply, when the PROBE comes back to its initiator, the initiator can declare the deadlock. An important feature of this algorithm is that only one process in a cycle will detect the deadlock, simplifying deadlock resolution.

Each node is given two labels: a **private label**, which although not constant is unique to the node at all times, and a **public label**, which can be sent to other processes and need not be unique. A process is represented as $(u/v)$ where $u$ and $v$ are respectively the public and private labels. For each process, initial private and public labels are identical. The WFG is maintained by the four state-transitions shown in Fig. 4, where $z = \mathbf{inc}(u)$ and function $\mathbf{inc}(u)$ yields a unique label greater than $u$. Labels not explicitly mentioned remain unchanged. The modification point from the algorithm in [7] is that the process's private label is incremented locally (without using the other process's label) when it is activated by receiving the ACK.

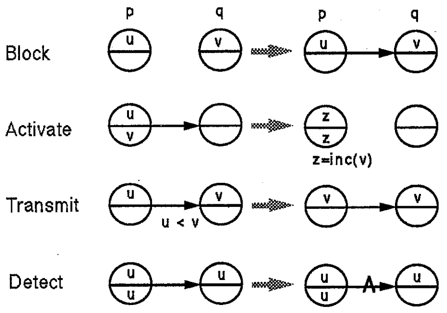

Figure 4: Deadlock detection algorithm

- **Block** creates an edge in the WFG if process $q$ receives a message from $p$ when its state is WAIT. Note that $q$ knows the public label of $p$ on receiving a message from $p$ which piggybacks it.
- **Activate** means that a process in WAIT state receives the ACK and changes its state to NORMAL. At that time, process $p$ increments its private label and also changes its public label to this new value. The private label of each node is always unique to that node and nondecreasing over time. These two properties can be achieved by representing labels as pairs of sequence numbers and process IDs, and $<$ can be chosen to be lexicographical ordering.
- **Transmit** propagates larger labels in the direction opposite the edges by transmitting a PROBE.
- **Detect** means that the PROBE with the private label of some process has made a whole round in a cycle, indicating a deadlock. Since process $q$ knows the private label of $p$ (which is equal to its public label) on receiving a message from $p$, $q$ can detect the deadlock. On detecting it, $q$ transmits NACK to $p$ and also transmits ACK to every other process $r$ in the cycle.

Figure 5 describes the instantaneous message passing (IMP) algorithm. Each process $p$ has a queue $Q_p$ for storing every message $m$ received during a WAIT state, as well as its sender's ID

$q$ and public label $public_q$. Each element of $Q_p$ is denoted as $(q : m, public_q)$. For a queue $Q$, a function **enq**$(Q,x)$ adds an element $x$ to the end of $Q$, a function **deq**$(Q)$ returns the first element of $Q$ and removes it from $Q$, and a function **del**$(Q,q)$ deletes the element with the sender's ID of $q$. For an element $e = (q : m, public_q)$, a function **mes**$(e)$ returns the message of $e$, $m$, and the function **ID**$(e)$ returns the sender's ID of $e$, $q$. A function **chgl**$(Q, q, newpub)$ changes the label of the element with the sender's ID of $q$ to $newpub$. The PROBE message carries the **plist**, which collects the process IDs in the cycle from the first receiver of the PROBE, and which will be used for transmitting the ACKs when the deadlock is detected. **plist**[i] indicates the $i$th process's ID of the plist. For the plist $L$, a function **add**$(L,x)$ adds an element $x$ to the end of $L$.

Further, the algorithm can be optimized for the deadlock cycle with only two processes. On transmitting a message $m$ to process $q$, process $p$ sets **waitfor** $:= q$. On receiving another message $(m' : public_q)$ from $q$ during a WAIT state, process $p$ executes **enq**$(Q_p, (q : m', public_q))$. Then if $waitfor = q$ and $public_p > public_q$, $p$ cancels the last $send_p$, delivers every message in $Q_p$, transmits $(ACK^*, p)$ to $q$ and $(ACK, p)$ to the other senders, and changes its state to NORMAL. On receiving $(ACK^*, p)$, $q$ delivers every message in $Q_q$ except that from $p$. This modification can reduce the number of system messages for the deadlock cycle with two processes.

## 4.2 Correctness proof

First, we will prove that this algorithm prevents message-crossing.

**Theorem 4.1** *If $H$ is any history using the above algorithm for message passing, then $H \in S(MP_c)$: i.e., there is no pair of messages $m$ and $m'$ in $H$ such that $(m \succ m') \wedge (m' \succ m)$.*

**Proof:** Let us assume that there is a pair of messages $m$ and $m'$ in $H$ such that $(m \succ m') \wedge (m' \succ m)$; i.e., there is a sequence of messages $MS : m = m_0, m_1, m_2, \ldots, m_{k-1} = m'$ such that $\forall i \in \{0,1,2,\ldots,k-1\} : m_i \succ_{p_i} m_{i+1} \pmod{k}$.

If every relation $m_i \succ_{p_i} m_{i+1}$ in $MS$ satisfies the $(-,+)$ relation of the exists-before relation (see Fig. 6 (a)), then there is a process $p_i$ with the maximum private label among the processes $p_j$ $(j = 0, 1, \ldots, k-1)$ in $MS$ whose sending message $m_i$ is cancelled by NACK.

If every relation $m_i \succ_{p_i} m_{i+1}$ in $MS$ is the $(+,-)$ relation, then $-m_0 \rightarrow +m_0 \rightarrow -m_1 \rightarrow \ldots \rightarrow -m_{k-1} \rightarrow +m_{k-1} \rightarrow -m_0$; this contradicts the irreflexiveness of the $\rightarrow$ relation.

Otherwise, $MP$ has a $(+,+)$ relation $+m_i \succ_{p_i} +m_{i+1}$ (see Fig. 6(b)). If you trace $MP$ in its direction from $m_{i+1}$, you will find the following two types of relations; $(-,+)$ relation $(-m_{i+1} \succ_{p_{i+1}} +m_{i+2})$ or $(-,-)$ relation. Let $m_j \succ_{p_j} m_{j+1}$ be the first $(-,-)$ relation $(-m_j \succ_{p_j} -m_{j+1})$ found by tracing from $m_{i+1}$. By further tracing $MP$, you will find the following two types of relations; $(+,-)$ relation $(+m_{j+1} \succ_{p_{j+1}} -m_{j+2})$ or $(+,+)$ relation. Let $m_k \succ_{p_k} m_{k+1}$ be the first $(+,+)$ relation $(+m_k \succ_{p_k} +m_{k+1})$ found by tracing from $m_{j+1}$. Since the algorithm prevent process $p_j$ from sending $m_{j+1}$ before receiving an ACK, which means $p_j$'s knowledge of the delivery of $m_{i+1}$, we get $delv_{p_i}(m_{i+1}) \rightarrow send_{p_j}(m_{j+1}) \rightarrow delv_{p_k}(m_{k+1})$. Tracing further from $m_{k+1}$ and using similar discussions, we can get $delv_{p_k}(m_{k+1}) \rightarrow delv_{p_i}(m_{i+1})$. Thus, $delv_{p_i}(m_{i+1}) \rightarrow delv_{p_i}(m_{i+1})$; a contradiction of the irreflexiveness of the $\rightarrow$ relation.

□

In a similar way to that in [7], we prove that this algorithm is free from deadlock. This proof is in Appendix A.

```
/* transmit message $m$ to $q$*/
if ( $state_p$ = NORMAL $\wedge$ $willsend_p(q; m)$) then
  { $\mathbf{trans}_p^q(m : public_p)$; $state_p$ := WAIT; }

/* receive message $m$ from $q$ */
On receiving message $(m : public_q)$ from $q$
if ($state_p$ = NORMAL) then
  { $\mathbf{deliver}_p(m)$; $\mathbf{trans}_p^q(ACK, \emptyset)$; }
if ($state_p$ = WAIT) then
  { $\mathbf{enq}(Q_p, (q : m, public_q))$;
    if ($public_p > public_q$) then
    { $\mathbf{trans}_p^q PROBE(public_p, \emptyset)$; $\mathbf{chgl}(Q_p, q, public_p)$); }
    if ($public_p = public_q$) then
    { $\mathbf{del}(Q_p, q)$; $\mathbf{trans}_p^q(NACK, plist[1])$;
      for every process $plist[i] (\neq p)$
          $\mathbf{trans}_p^{plist[i]}(ACK, plist[i+1])$ ;
      $\mathbf{trans}_p^p(ACK, \emptyset)$ ; }
  }

/* receive ACK */
On receiving $(ACK, plist[i])$
while ($Q_p \neq \emptyset$)
  { $e := \mathbf{deq}(Q_p)$; $q := \mathbf{ID}(e)$; $\mathbf{delv}_p(\mathbf{mes}(e))$;
    if($q \neq plist[i]$) then $\mathbf{trans}_p^q(ACK, \emptyset)$ ; }
$state_p$ := NORMAL; $private_p := \mathbf{inc}(private_p)$;
$public_p := private_p$;

/* receive NACK */
On receiving $(NACK, plist[1])$
cancel the last $send_p$;
while ($Q_p \neq \emptyset$)
  { $e := \mathbf{deq}(Q_p)$; $q := \mathbf{ID}(e)$; $\mathbf{delv}_p(\mathbf{mes}(e))$;
    if ($q \neq plist[1]$) then $\mathbf{trans}_p^q(ACK, \emptyset)$ ; }
$state_p$ := NORMAL;

/* receive PROBE for detecting deadlock */
On receiving $PROBE(public_q, plist)$ from $q$
$public_p := public_q$; $\mathbf{add}(plist, p)$;
for every sender $r$ of the element in $Q_p$
  { if ($public_p > public_r$) then
    { $\mathbf{trans}_p^r PROBE(public_p, plist)$; $\mathbf{chgl}(Q_p, r, public_p)$; }
    if ($public_p = public_r$) then
    { $\mathbf{del}(Q_p, r)$; $\mathbf{trans}_p^r(NACK, plist[1])$;
      for every process $plist[i] (\neq p)$
          $\mathbf{trans}_p^{plist[i]}(ACK, plist[i+1])$ ;
      $\mathbf{trans}_p^p(ACK, \emptyset)$ ; }
  }
```

Figure 5: IMP algorithm for control process $p$

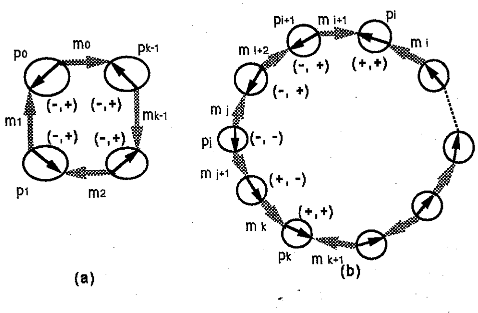

Figure 6: Cases of message-crossing

**Theorem 4.2** *If a cycle of $k$ nodes forms and persists long enough, exactly one of these nodes will execute the Detect step of the algorithm. This will happen after $k-1$ consecutive Transmit steps.*

Further, we can prove that only genuine deadlock is detected in the same way as that in [4].

**Theorem 4.3** *If a deadlock is detected, a cycle of blocked nodes exists.*

Finally, we will prove that the algorithm satisfies the fairness condition.

**Theorem 4.4** *The above algorithm satisfies the fairness condition.*

**Proof:** If the edge from $p$ to $q$ is in a deadlock cycle and $private_p$ is the largest private label in it, then the send event of $p$ to $q$ is cancelled. On detecting the cycle, though the private label of $p$ remains the same value, those of the other processes $r$ in the cycle - including $q$ - are increased by a function $inc(r)$. Thus, even if the further trials of send from $p$ to $q$ make deadlock cycles, the private label of $p$ will eventually become smaller than that of $q$, and $p$ will be able to send a message to $q$.

□

### 4.3 Complexity

We consider here the message complexity, time complexity, and concurrency of the above algorithm.

- **Message Complexity:** As the *message complexity* $C_M(A)$ of an algorithm $A$, we will use the maximum number of additional system messages transferred to send $k$ user messages in a deadlock cycle. Since at most $k(k-1)/2$ PROBE messages, $(k-1)$ ACK messages, and one NACK message are required, the message complexity of the IMP algorithm, $C_M(IMP)$, is $k(k+1)/2$.
- **Time Complexity:** As the *time complexity* $C_T(A)$ of an algorithm $A$, we will use the maximum interval of a WAIT state (i.e., the maximal interval from a transmit event of user message to the instance the successive transmit event can occur) when the upper bound on interprocess communication delay can be assumed to be $T$. The time complexity of the IMP algorithm, $C_T(IMP)$, is $2kT$, where $k$ is the number of messages in a deadlock cycle. This is because the delay between the first transmit event and the last receive event of the $k$ messages in the deadlock cycle is bounded by $kT$, the delay between the last receive and Detect event is bounded by $(k-1)T$, and the delay between the Detect event and the receive event of NACK is bounded by $T$.
- **Concurrency:** As the measurement of the *concurrency* $C_C(A)$ of an algorithm $A$, we will use the maximum number of user messages concurrently transferred in an $n$-process system. It is clear that the concurrency of the IMP algorithm, $C_C(IMP)$, is equal to $n-1$. This is higher than that of every algorithm for the rendezvous REN since $C_C(REN) = n/2$ ([1] etc.).

## 5 Application and Discussions

The instantaneous message passing is applicable as a mechanism for resolving process collision[6] in communication protocols. A major concern of this mechanism is its run-time overhead. In this section, we use a simple example to demonstrate that some overhead will be required no matter how asynchronism is resolved in a protocol, and that the overhead of the IMP algorithm is reasonable.

Traditionally, coordination errors in communication protocols are resolved during the design phase. Figure 7(a), adapted from [11], is the simplified connection management protocol specification. Two processes $p$ and $q$ use messages $EST$ and $TRM$ for requesting to establish and to terminate a connection. When processes $p$ and $q$ simultaneously send $EST$ to each other, collision occurs and both processes will receive an $EST$ at state $S_2$. However, since the reception of $EST$ is not specified at $S_2$, this represents an error in this protocol (*unspecified reception*). A modification, also from [11], is shown in Fig.7(b). In this modification, the unspecified reception error is resolved in favor of $p$ by adding extra states and transitions, which represent a form of overhead. In this new specification, if a collision does not occur, an $EST$ and an $ACK$ will be sent in order to establish a connection; otherwise, two $EST$ messages and an $ACK$ will be exchanged.

In contrast, when the IMP algorithm is used, an $EST$ and an $ACK$ will be transmitted if a collision does not occur; otherwise, two $EST$ messages, one PROBE message, and one $NACK$ will be exchanged (an $ACK$ message from itself may be neglected). If the algorithm is optimized for the deadlock cycle with two processes, then two $EST$ messages and an $ACK$ will be transmitted; the number of messages exchanged using the IMP mechanism is the same as that using the traditional approach.

With the traditional approach, design modification is difficult, and the modified design may be more complicated and difficult to understand. It is also expected that in most cases only a few process might constitute a deadlock cycle. Though cases with three or more processes in a deadlock cycle must be further studied, with these facts in mind, it seems that the run-time overhead of the IMP mechanism is reasonable.

A similar approach has been proposed as a *self-synchronizing communication protocol* (SSCP), which uses each specification explicitly[6]. The IMP mechanism is superior to this approach in the following points: (1) concurrency (in the case of deadlock, at most one request message is selected and the others are cancelled in SSCP), and (2) memory requirement (since SSCP needs backtracking which cancels plural send events, each process needs keep its trace in its memory).

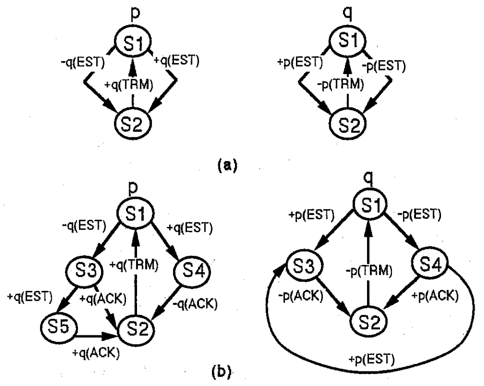

Figure 7: Counterexamples

## 6 Conclusions

This paper presents a mechanism for *simulating* instantaneous message passing in asynchronous systems. Section 3 shows that no message-crossing is *indistinguishable from* instantaneous message passing, and clarifies that it achieves higher synchronization than the causal ordering. This section also clarifies its usefulness for internal specifications. Section 4 shows an algorithm to achieve no message-crossing. This algorithm allows an $n$-process system to have $n-1$ concurrent messages, and it satisfies deadlock-freeness and fairness conditions. This algorithm also requires only one ACK for each user message if there is no possibility of deadlock. When deadlock occurs with $k$ user messages in a deadlock cycle, $k(k+1)/2$ additional messages are required in the worst case; the order of this number is equal to the best known message complexity of deadlock detection algorithm. Furthermore, the maximal interval of a WAIT state is bounded by $2kT$, where $T$ is the assumed upper bound on interprocess communication delay.

The implemental study of the instantaneous message passing, which combines it with a time-out method by considering statistical data of message transmission delay and the number of chained messages, remains for further study.

**Acknowledgements**

I would like to thank Sam Toueg, Ajei Gopal, and Tushar Chandra in Cornell University for discussing in the earlier stage of this work, and also Yoshifumi Manabe and Haruhisa Ichikawa for discussing this work and commenting on earlier drafts of this paper.

## A Proof of Theorem 4.2

**Lemma A.1** *If there is an edge from $p$ to $q$ in WFG as in $p(\frac{u}{v}) \longrightarrow q(\frac{w}{x})$ and $u > w$, then $u = v$.*

**Proof:** The definitions of the Activate, Detect, and Block steps guarantee that this will be true when an edge first forms. The only way the label $u$ will change during the lifetime of that edge is if a Transmit step is executed, and that will not happen as long as $u > w$ and $w$ is nondecreasing during the life time of that edge. This is because that edge will have been killed by the Activate step before the label $w$ will be changed by the Activate step to another edge incident from $q$.

□

**Lemma A.2** *The maximum public label value in a cycle is equal to the private label of one and only one node in that cycle.*

**Proof:** Unless all the public labels have the maximum value when the last receive event occurs to form a cycle, at least one node with maximum public value must precede a node with a lower public label. Thus by Lemma A.1, the private label of the preceding node must be the same as the maximum public label value. Since no Activate operation can be performed by nodes in a cycle, this will remain true throughout its lifetime. Otherwise, it is clear that the private label of the sender of the last receive message is equal to the maximum public label. Private labels are unique, so only one node can obey this condition.

□

Thus, the proof of Theorem 4.2 is clear. If a cycle forms, $k-1$ Transmit steps will carry the largest public label value all the way around the cycle. By Lemma A.2, this means one and only one node will eventually execute the Detect step.

## References

[1] R. Bagrodia, "Synchronization of asynchronous processes in CSP," ACM Trans. on Programming Languages and Systems, Vol. 11, No. 4, pp.585-597 (1989).

[2] K. Birman and T. Joseph, "Reliable communications in presence of failures", ACM Trans. on Computer Systems, Vol. 5, No. 1, pp. 47-76 (1987).

[3] C.A.R. Hoare, "Communicating sequential processes," Commun. of ACM, Vol. 21, No. 8, pp. 666-677 (1978).

[4] E. Knapp, "Deadlock detection in distributed databases," ACM Computing Surveys, Vol. 19, No. 4, pp.303-328 (1988).

[5] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," Commun. of the ACM, Vol. 21, No. 7, pp. 558-565 (1978).

[6] H. P. Lin and H. E. Stovall, III, "Self-synchronizing communication protocols," IEEE Trans. on Computers, Vol. 38, No. 5, pp. 609-625 (1989).

[7] D.P. Mitchell and M. J. Merritt, "A distributed algorithm for deadlock detection and resolution," Proc. of the 3rd ACM Symp. on Principles of Distributed Computing, pp.282-284 (1984).

[8] G. Neiger and S. Toueg, "Simulating synchronized clocks and common knowledge in distributed systems," Technical Report, Dept. of Computer Science, Cornell University, TR90-1086 (1990). To appear in the Journal of the ACM.

[9] A. Schiper, J. Eggli, and A. Sandoz, "A new algorithm to implement causal ordering," Lecture Notes in Computer Science 392, Distributed Algorithms, pp.219-232 (1989).

[10] M. Singhal, "Deadlock detection in distributed systems," IEEE Computer, Vol. 22, No. 11, pp.37-48 (1989).

[11] P. Zafiropulo, "Protocol validation by duologue-matrix analysis," IEEE Trans. Commun., Vol. COM-26, No. 8, pp. 1187-1194 (1978).