# 非同期分散システムにおける瞬時メッセージ転送－その2－

曽根岡 昭直　　　　茨木 俊秀
NTT ソフトウェア研究所　京都大学工学部

**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未知）に起因する難しさがある。通信遅延を意識せずに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、本稿では瞬時にメッセージ転送がなされているように各プロセスに見せかける転送方式を提案する。本方式は以下の性質を持つ。1) 各プロセスがクライアントともサーバとも動作するパートナモデルにおいても、デッドロック状態に陥ることなく適用できる。2) 1つのメッセージを転送するのに3つの信号を要し、メッセージ複雑度準最小（下界より高々1大きいだけ）である。3) 時間複雑度を送信要求から対応する送信イベントの生起までの最悪遅延で評価した場合、最適な時間複雑度を持つ。さらに、本稿では、ランダム化技術を用いることにより、より短い平均遅延を達成するアルゴリズムも提案する。

# Instantaneous Message Passing in Asynchronous Distributed Systems -II-

**Terunao SONEOKA**　**Toshihide IBARAKI**
NTT Software Laboratories　Kyoto University
Musashino-shi, Tokyo 180　Kyoto-shi, Kyoto 606

**Abstract**　Asynchrony (unknown message transmission delay) complicates the design of protocols for distributed systems. To simplify the protocol design task, therefore, this paper proposes an interprocess communication mechanism *simulated* by instantaneous message passing. This mechanism has the following properties. 1) It is applicable without deadlock to the *partner model* in which each process acts as both client and server. 2) It requires at most three signals to send a message, which is shown to be the quasioptimum message complexity (at most one larger the lower bound). 3) It has the optimal time complexity, where time complexity is evaluated by the worst-case delay from a send request to the occurrence of the corresponding send event under the assumption that an upper bound is known for the interprocess communication delay. Furthermore, a modified algorithm is proposed for attaining a shorter average delay by using a randomization technique.

## 1 Introduction

Distributed systems without exact knowledge 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 $C$ the progress of job $X$ by sending message $m_3$. However, when $C$ receives $m_3$, as $C$ has not yet received $m_1$, $C$ is confused about what $B$ is asking. This situation corresponds to the violation of the *causal delivery* in [2, 12] (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 $m_2$ "Let's 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 *collision* in [4] [1] (see Fig. 1(b)).
- A monitoring process $A$ is looking for the process having a token. If $A$ asks $B$ and $C$ "Do you have a token?" by sending messages $m_1$ and $m_2$ while the token is in transit from $B$ to $C$, then $A$ receives "No" from both of them; so $A$ might mistakenly assume that the token has been lost (see Fig. 1(c)).

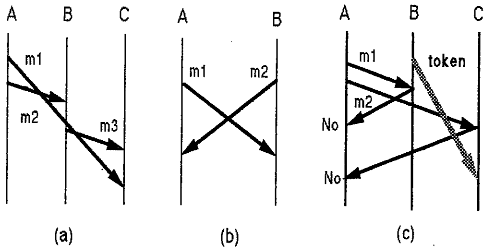


Figure 1: Examples of coordination errors in asynchronous distributed systems

All of these errors can be easily avoided if every message passing is achieved instantaneously. Unfortunately, one cannot implement a system with instantaneous message passing. Along the lines of [11], which proposed a mechanism simulated by synchronized clocks in distributed systems in order to simplify the design task, this paper[2] proposes a mechanism *simulated* by instantaneous message passing (i.e., synchronous message passing) in distributed systems. Here, *simulated* means that each process in a system can perform as if the system is using an ideal instantaneous message passing.

[1]The collision is very often found in OSI protocols and telephone service protocols.

[2]This paper is a revised version of [13]. The paper [13] proposes an inefficient algorithm whose message complexity is $O(k^2)$, where $k$ is the maximum number of messages in a deadlock cycle.

This paper first defines the concept of a system "*simulated*" by another system with a different message passing property, and presents a necessary and sufficient condition for simulated by instantaneous message passing. This condition enables us to devise more efficient means than the totally ordered message passing using Lamport's mutual exclusion algorithm[10] and Bagrodia's *rendezvous* algorithm[1] for the *generalized alternative command* suggested in CSP[9]. It is also shown that the proposed mechanism simulated by instantaneous message passing achieves higher (stricter) synchrony than the causal delivery in [2, 12]. We then propose a naive algorithm for achieving the mechanism. This algorithm is deadlock-free even for the *partner model* of distributed systems, where each process acts as both client and server. The partner model is important for communications software such as caller/callee processes in switching programs and SCP/SSP entities in an Intelligent Network. It requires at worst three signals for transmitting a message, which is shown to be quasioptimum in message complexity (at most one larger than the lower bound). The naive algorithm is further shown to have the optimal time complexity: that is, the worst-case delay from a send request to the occurrence of the corresponding send event is bounded by $2kT$, where $k$ is the maximum length of the dependent message sequence and $T$ is assumed to be an upper bound on interprocess communication delay.

Among the related communication mechanisms for asynchronous distributed systems are the *remote procedure call* (RPC)[3], Bagrodia's rendezvous algorithm[1], and Goldman's synchronous multicast[6]. Although the RPC is widely used for the *client/server model*, it may cause deadlocks in the partner model. The instantaneous message passing mechanism proposed in this paper is applicable without deadlock even for the partner model. In Bagrodia's rendezvous algorithm, on the other hand, the sender and receiver must both be ready before message passing execution. Thus, the concurrency of the redezvous algorithm[1] is low in the sense that at most $\frac{n}{2}$ processes can send messages concurrently in an $n$-process system. Although Goldman's synchronous multicast is similar to our algorithm, his algorithm requires 3 or 4 signals to handle a message[3] and the worst-case delay from a send request to the occurrence of the corresponding send event is equal to $4(k+1)T$.

The rest of the paper is organized as follows. Section 2 presents a model of distributed systems and its related definitions. Section 3 first presents a necessary and sufficient condition for simulated by instantaneous message passing, which enables us to devise more efficient algorithms than the previously proposed ones. Next the effectiveness of the instantaneous message passing is shown. Section 4 first proposes a naive algorithm for this mechanism and proves its correctness, freedom from deadlock, and fairness. Then the lower bounds of message complexity and time complexity are derived for instantaneous message passing problem. From these, the proposed algorithm is shown to be quasioptimum in message complexity and optimum in time complexity. Section 5 presents a modified algorithm for attaining a shorter average delay by using a randomization technique.

## 2 Model and definitions

### 2.1 Model

We consider an *asynchronous* distributed system consisting of two layers: the *application layer* consisting of $n$ user processes and the *message-passing layer* with channels, through which user processes communicate with each other. By asynchronous, we mean that there is (1) no global clock (each user process is not

[3]The 4th signal is necessary for avoiding deadlocks, and can be shown to be necessary even for single-cast cases.

aware of the physical time, though it has its own local clock), (2) no knowledge about the message transmission delay, and (3) no assumption about the relative speed of user processes. We however assume that (1) user processes and message-passing layer are reliable, i.e., every message sent out is eventually received by the destination user process after some nonnegative time has elapsed [4] if measured along the physical time axis, and that (2) each user process has a distinct ID. For simplicity, we let $P = \{u_1, \ldots, u_n\}$ be the set of user process ID's in the system.

In order to send a message $m$ to another user process $u_q$, user process $u_p$ issues "*will-send*$_p^q(m)$" to the message-passing layer. Upon receiving it, the message-passing layer sends $m$ through the channel and notifies $u_p$ of the occurrence of *send event* (denoted by $send_p^q(m)$ or, shortly, $send_p(m)$) if some condition is satisfied. After a short delay, the message-passing layer delivers $m$ to the destination user process $u_q$ (denoted by $delv_q^p(m)$ or, shortly, $delv_q(m)$). Besides *send event* and *deliver event*, user processes also execute *internal event*, which is the result of actions autonomously taken by it. We assume for simplicity that each event takes exactly one local time unit to execute.

### 2.2 History

To define "simulated", the concept of history must be clearly defined. We will follow the notation used by Neiger and Toueg[11] to describe the *history* of a system, which is a specific execution of a system. A history $H$ consists of the following four history functions; $H =< C, Q, A, MP >$. A *system S* is identified by the set of histories that correspond to all possible executions of the system and is denoted by $S(C, Q, A, MP)$.

- The *clock history function* $C$ maps from user processes $P$ and physical time space $R$ (non-negative real numbers) to clock time space $N$ (natural numbers); i.e., $C : P \times R \mapsto N$. $C(u_p, t)$ is the time on $u_p$'s clock at physical time $t$. Since a user process clock never decreases, all clock history functions satisfy the condition: $CC : \forall u_p \in P\ \forall t_1, t_2 \in R\ [t_1 < t_2 \Rightarrow C(u_p, t_1) \leq C(u_p, t_2)]$.
- The *state history function* $Q$ maps from user processes and clock times to user process states $S$; i.e., $Q : P \times N \mapsto S$. User process $u_p$ is in state $Q(u_p, c)$ when its clock shows $c$.
- The *event history function* $A$ maps from user processes and clock times to events $E$; i.e., $A : P \times N \mapsto E$. Note that only a single event can occur in a user process $u_p$ at a clock time. This is a partial function, since a user process may not have an event at every instant of clock time. User process $u_p$ has event $A(u_p, c)$ when its clock shows $c$.
- The *message-passing history function* $MP$ maps from pairs of send events and the corresponding delivery events $M \subset E \times E$ to pairs of physical times; i.e., $MP : M \mapsto R \times R$. A message $m \in M$ is sent at $r_1$ time and delivered at $r_2$ time if $MP(m) = (r_1, r_2)$. Since the message transmission delay is a nonnegative time, for any $m \in M$, $r_1 \leq r_2$ if $MP(m) = (r_1, r_2)$.

Two histories $H_1 =< C_1, Q_1, A_1, MP_1 >$ and $H_2 =< C_2, Q_2, A_2, MP_2 >$ are *equivalent to user process* $u_p$, denoted by $H_1 \overset{p}{\sim} H_2$, if $Q_1(u_p, c) = Q_2(u_p, c)$ and $A_1(u_p, c) = A_2(u_p, c)$ for any $c \in N$. Furthermore, two histories $H_1$ and $H_2$ are *equivalent*, denoted by $H_1 \sim H_2$, if $H_1 \overset{p}{\sim} H_2$ for all $u_p \in P$; otherwise, we denote $H_1 \not\sim H_2$. Informally, $H_1 \sim H_2$ if all user processes behave in exactly the same manner according to their local clocks in both histories. Since user processes cannot observe physical time (they can only observe their local clocks), they cannot distinguish $H_1$ from $H_2$.

[4]Note that we do *not* assume that channels are FIFO (first-in-first-out).

### 2.3 Relations between events and messages

The *"happens-before"* relation "$\rightarrow$" is defined on the set of events of a system by Lamport[10] 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 user process $u_p$, 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*).

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

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

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

In particular, we denote $m_1 \prec_p m_2$ iff one of the above conditions 1, 2, 3, or 4 holds for a given user process $u_p$.

## 3 Simulating instantaneous message passing

### 3.1 Necessary and sufficient condition

This section first defines the concept of a system ***"simulated"*** by another system with a different message-passing property, and presents a necessary and sufficient condition for simulated by instantaneous message passing. We consider systems with different message passing : a system using a message passing layer with property $MP$ is simply denoted by $S(MP)$ by omitting $C$, $Q$, and $A$ if the systems with the same $C$, $Q$, and $A$ are considered. To clarify the relationship between a system $S(MP_1)$ with a message passing property $MP_1$ and a system $S(MP_2)$ with a message passing property $MP_2$, we define the following terminology: $S(MP_1)$ is *simulated* by $S(MP_2)$, denoted by $S(MP_1) \trianglelefteq S(MP_2)$, (or simply, $MP_1$ is simulated by $MP_2$) 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 $\exists H \in S(MP_1)\ \forall H' \in S(MP_2)\ [H \not\sim H']$.

The following lemma holds.

**Lemma 3.1** *Relation $\trianglelefteq$ is transitive, i.e., $S(MP_1) \trianglelefteq S(MP_2) \wedge S(MP_2) \trianglelefteq S(MP_3) \Longrightarrow S(MP_1) \trianglelefteq S(MP_3)$.*

**Proof:**
Obvious because $\forall H \in S(MP_1) \exists H'' \in S(MP_3)[H \sim H'']$ if $\forall H \in S(MP_1) \exists H' \in S(MP_2)[H \sim H']$ and $\forall H' \in S(MP_2) \exists H'' \in S(MP_3)[H' \sim H'']$.

□

We will consider the following ideal instantaneous message passing property $I$, which cannot be achieved by asynchronous distributed systems in the real world.

$I$: **Instantaneous message passing property.** For any message $m$, its transmission delay is always equal to zero; that is, every message is transmitted instantaneously.

The following actual message-passing properties $CD$, $NMC$, $RDV$, and $TO$ are also considered. The algorithms for achieving causal delivery property ($CD$) in asynchronous distributed systems are proposed in [2, 12]. The algorithms for no-message-crossing property ($NMC$) will be proposed in the following sections. An algorithm for rendezvous property ($RDV$) is proposed in [1] and an algorithm for total ordering property ($TO$) can be easily obtained by using Lamport's mutual exclusion algorithm[10].

$CD$: **Causal delivery property.** If $send_p^r(m) \rightarrow send_q^r(m')$, then $delv_r^p(m) \xrightarrow{r} delv_r^q(m')$.

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

$RDV$: **Rendezvous property.** For any message $m$ sent by $u_p$, the receiver $u_q$ does not send or deliver any other messages $m'$ in the physical-time interval between $send_p^q(m)$ and $delv_q^p(m)$.

$TO$: **Total ordering property.** There is at most one message in transit in the system at any physical time.

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

**Theorem 3.1** *1. $S(MP) \trianglelefteq S(I)$ iff $S(MP) = S(NMC)$.*

*2. $S(NMC) \subseteq S(CD)$. $S(CD) \not\trianglelefteq S(NMC)$. $S(CD) \not\trianglelefteq S(I)$.*

*3. $S(RDV) \subseteq S(NMC)$ (hence $S(RDV) \trianglelefteq S(NMC)$). $S(NMC) \not\subseteq S(RDV)$. $S(RDV) \trianglelefteq S(I)$.*

*4. $S(TO) \subseteq S(RDV)$ (hence $S(TO) \trianglelefteq S(RDV)$). $S(RDV) \not\subseteq S(TO)$. $S(TO) \trianglelefteq S(I)$.*

**Proof:**

1. Consider any $H \in S(NMC)$, where every message can be partially ordered according to the relation $m \prec m'$. Let us assign any physical time $t_i \in R$ to message $m_i$ such that $m_i \prec m_j \Rightarrow t_i < t_j$. From this, it is clear that the clock of any user process $u_p$, $C(u_p, t)$, satisfies the clock condition $CC$; that is, $t_1 < t_2 \Rightarrow C(u_p, t_1) \leq C(u_p, t_2)$. Thus, there exists $H' \in S(I)$ such that $H \sim H'$. On the other hand, if $S(MP)$ does not satisfy $NMC$, then there is a pair of messages $m$ and $m'$ in $H'' \in S(MP)$ such that $(m \prec m') \wedge (m' \prec m)$. Let us assume that $\exists H' \in S(I)[H'' \sim H']$ and respectively assign any physical times $t$ and $t' \in R$ to messages $m$ and $m'$, then we get $t < t'$ and $t' < t$; a contradiction.

2. The first predicate is trivial. On the other hand, Fig. 1(b) and (c) are counterexamples of $S(CD) \trianglelefteq S(NMC)$. The third one is straightforward from Lemma 3.1.

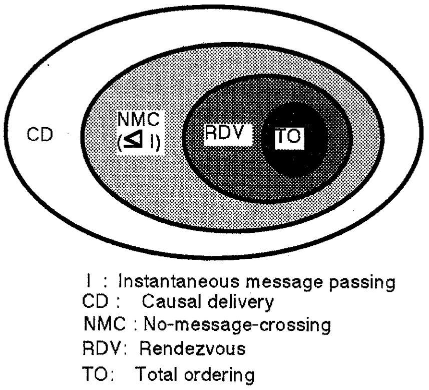


Figure 2: Relationships among synchronous message-passing categories

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

   Case 1: Every relation $m_i \prec_{p_i} m_{i+1}$ in $MS$ is the $(+,-)$ relation of the exists-before relation.
   In this case, $send(m_0) \rightarrow delv(m_0) \xrightarrow{p_1} send(m_1) \rightarrow \ldots \xrightarrow{p_{k-1}} send(m_{k-1}) \rightarrow delv(m_{k-1}) \xrightarrow{p_0} send(m_0)$ holds; this contradicts the irreflexiveness of the $\rightarrow$ relation.

   Case 2: Every relation $m_i \prec_{p_i} m_{i+1}$ in $MS$ is the $(-,+)$ relation (see Fig. 3 (a)).
   Since the receiver of a message $m$ does not send any other messages $m'$ in the physical-time interval between $send(m)$ and $delv(m)$ in $H \in S(RDV)$, $send(m_i)$ occurs before $send(m_{i+1})$ $(i = 0, \ldots, k-2)$ in physical-time space. Thus $send(m_{k-1})$ occurs at the receiver of $m_0$, $u_{p_{k-1}}$, between $send(m_0)$ and $delv(m_0)$ in physical-time space; this is a contradiction.

   Case 3: $MS$ has a $(+,+)$ relation $+m_i \prec_{p_i} +m_{i+1}$ (see Fig. 3(b)).
   If you trace $MS$ from $m_{i+1}$, you will find either of the following two types of relations; the $(-,+)$ relation $(-m_{i+1} \prec_{p_{i+1}} +m_{i+2})$ or the $(-,-)$ relation. Let $m_j \prec_{p_j} m_{j+1}$ be the first $(-,-)$ relation $(-m_j \prec_{p_j} -m_{j+1})$ found by tracing $MS$ from $m_{i+1}$. By further tracing $MS$, you will find either of the following two types of relations; the $(+,-)$ relation $(+m_{j+1} \prec_{p_{j+1}} -m_{j+2})$ or the $(+,+)$ relation. Let $m_k \prec_{p_k} m_{k+1}$ be the first $(+,+)$ relation $(+m_k \prec_{p_k} +m_{k+1})$ found by tracing $MS$ from $m_{j+1}$.
   Since the receiver of a message $m$ does not send or deliver any other messages $m'$ in the physical-time interval between $send(m)$ and $delv(m)$ in $H \in S(RDV)$, $delv(m_i)$ occurs before $send(m_{i+1})$, $send(m_{i+1})$ occurs before $send(m_{i+2}), \ldots$, $send(m_{j-1})$ occurs before $send(m_j)$ in physical-time space. From this and the causality from $send(m_j)$ and $delv(m_k)$, $delv(m_i)$ occurs before $delv(m_k)$ in physical-time space. Tracing further from $m_{k+1}$ and using similar discussions, it can be derived that $delv(m_k)$ occurs before $delv(m_i)$ in physical-time space. Thus, $delv(m_i)$ occurs before $delv(m_i)$ in physical-time space; a contradiction.

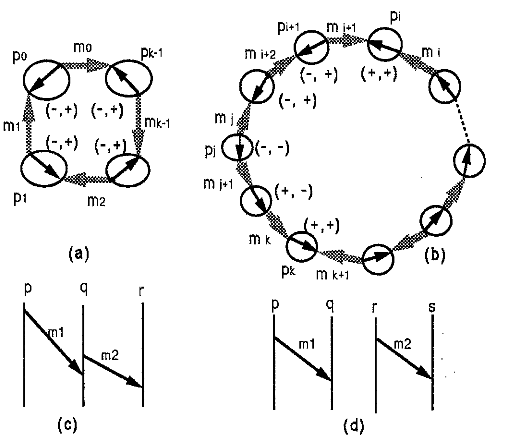


Figure 3: Explanation for proof of Theorem 3.1

On the other hand, Fig. 3(c) is a counterexample of $S(NMC) \subseteq S(RDV)$. The third one is straightforward from Lemma 3.1.

4. The first predicate is trivial. On the other hand, Fig. 3(d) is a counterexample of $S(RDV) \subseteq S(TO)$. The third one is straightforward from Lemma 3.1.

□

The first predicate of this theorem shows that no-message-crossing property $NMC$ is necessary and sufficient for simulated by instantaneous message passing. The second relation shows that the no-message-crossing property (i.e., instantaneous message passing property) achieves higher (stricter) synchronization than the causal delivery property. The third and fourth relations show that the rendezvous property and the total ordering property are too strong for simulated by instantaneous message passing.

### 3.2 Effectiveness of instantaneous message passing

Before explaining the effectiveness of instantaneous message passing, we prepare the following concepts about the application protocols. Each user process $u_p$ runs a *local application protocol* $(\delta_{u_p}, \Pi_{u_p})$. $\delta_{u_p}$ determines the next state of the user process from its present state and the event executed; i.e., $\delta_{u_p} : S \times E \mapsto S$. If $u_p$ is in state $s$ and event $a$ occurs, then it changes to state $\delta_{u_p}(s, a)$. Given $u_p$'s state, $\Pi_{u_p}$ specifies the next event candidates executed by $u_p$; i.e., $\Pi_{u_p} : S \mapsto 2^E$. If $u_p$ is in state $s$, then it executes an event in $\Pi_{u_p}(s)$. If $a \in \Pi_{u_p}(s)$ and $a$ is an internal event, then $|\Pi_{u_p}(s)| = 1$; that is, internal events are deterministically selected. However, user process $u_p$ is allowed to nondeterministically select one of several communication (i.e., non-internal) event candidates, e.g., send events and deliver events for execution, as adopted in some of the existing local application protocols written with the generalized alternative command for CSP. The set of local application protocols for all user processes, $\Pi = \{(\delta_{u_p}, \Pi_{u_p}) | u_p \in P\}$, is called an *application protocol*.

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

$$\forall u_p \in P \; \forall c \in N \quad [\; A(u_p, c) \text{ is defined} \Rightarrow Q(u_p, c+1) = \delta_{u_p}(Q(u_p, c), A(u_p, c)) \wedge A(u_p, c) \in \Pi_{u_p}(Q(u_p, c))].$$

Conceptually, histories consistent with $\Pi$ are those that may result when we run $\Pi$ in the system. This definition leads to the following[11]:

**Lemma 3.2** *If $H_1 \sim H_2$ and $H_1$ is consistent with application protocol $\Pi$, then $H_2$ is also consistent with $\Pi$.*

We consider that any problem to be solved in a distributed system $S$ is specified by a predicate $\Sigma$ on histories. Furthermore, we assume that the specification $\Sigma$ is *internal* in the sense that, for any equivalent histories $H_1 \sim H_2$, $H_1$ satisfies $\Sigma$ if and only if so does $H_2$. This in particular implies that $\Sigma$ does not contain the concept of physical time $t$. Many problems in distributed systems can be defined without involving $t$; for example, the concept of serializability in executing transactions. We say that a protocol $\Pi$ solves the problem specified by $\Sigma$ if any history consistent with $\Pi$ in $S$ satisfies $\Sigma$.

Now we explain the effectiveness of instantaneous message passing. Suppose that a designer verifies an application protocol for a system with an ideal instantaneous message passing property $I$. That is, the designer proves that all histories in $S(I)$ consistent with the application protocol under consideration satisfy a given specification $\Sigma$. Then, the application protocol may not satisfy $\Sigma$ if it is run in a different system that does not have $I$. The following theorem shows that if $\Sigma$ is internal and if a system uses the message passing $MP$ simulated by $I$ (i.e., $S(MP) \trianglelefteq S(I)$), then the application protocol remains correct.

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

**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 simulated by $I$, there is an $H' \in S(I)$ such that $H' \sim H$. By Lemma 3.2, $H'$ is also consistent with $\Pi$. By the assumption, $H'$ satisfies $\Sigma$. Since $\Sigma$ is internal and $H' \sim H$, $H$ also satisfies $\Sigma$.

□

This property simplifies the design and verification of application protocols. For example, if an application protocol is run on a system with the property $NMC$, which is simulated by instantaneous message passing property $I$, then it is sufficient to design and verify under the assumption of instantaneous message passing $I$, and the coordination errors illustrated in Fig. 1 can be easily resolved even without the assumption of property $I$. Traditionally, coordination errors have been resolved during the design phase by adding extra states and transitions; in some cases, however, the modification was difficult and error-prone, and the modified design was also more complicated and difficult to understand. The instantaneous message passing property simplifies these points.

## 4 Naive algorithm for achieving no-message-crossing property

We will consider the message-passing layer of a system consisting of *control processes* and channels, and propose a naive algorithm for a control process achieving no-message-crossing property $NMC$, which by Theorem 3.1 is simulated by instantaneous message passing $I$.

### 4.1 Algorithm

The system architecture is illustrated in Fig. 4, where $u_p$ denotes the user process engaged in an application protocol and $p$ denotes

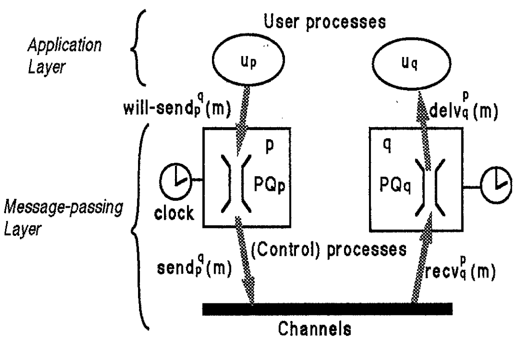


Figure 4: System architecture

the control process working for $u_p$. User process $u_p$ and its control process $p$ are located at the same site and communicate with each other through shared memory. Each control process $p$ has a distinct ID and maintains a pending queue $PQ_p$ for storing messages until they are sent or delivered. Upon receiving "*will-send$_p^q$(m)*" from a user process $u_p$, the control process $p$ sends $m$ to another control process $q$ through the channel and notifies $u_p$ of the occurrence of *send event* (denoted by $send_p^q(m)$ or, shortly, $send_p(m)$) if some condition is satisfied. On receiving the message $m$ (denoted by $recv_q^p(m)$ or, shortly, $recv_q(m)$), $q$ delivers $m$ to its user process $u_q$ (denoted by $delv_q^p(m)$ or, shortly, $delv_q(m)$) if some condition is satisfied. On delivery of a message $m$, the receiver user process $u_q$ changes its state, and may decide to withdraw some messages $m'$ issued to its control process $q$ with "*will-send(m')*" but not yet sent out.

Each control process $p$ has a clock, and can issue timestamps $T_p$ based on it. To sever ties between the same clock time of different control processes, each control process $p$ issues a timestamp $T_p$ of its clock time $C_p$ appended with its process ID, $p$, i.e., $T_p = (C_p, p)$. The timestamped message $m$ is denoted by "$T_p$: m". The ordering $<$ among timestamps can be chosen to be lexicographical ordering. Control process $p$ increments its clock $C_p$ at each local event, and at most one event can occur in a control process $p$ at the same clock time. Upon receiving a timestamped message "$T_q$ : $m$" from control process $q$ where $T_q = (C_q, q)$, control process $p$ always sets its clock to: $C_p' = \max\{C_p, C_q\} + 1$ where $C_p$ is the current clock value of $p$. The pending queue $PQ_p$ stores messages in the order of increasing timestamps. Hereafter, as the message-passing layer of a system is mainly considered, control processes are simply called processes.

The naive algorithm of process $p$ given in Fig. 5 achieves no-message-crossing property $NMC$ in three phases[5] as shown in Fig. 6; this is inspired by the deadlock-free concurrency control algorithm for distributed database systems[8]. Here, $LT_p$ is max {the timestamp of the latest message sent from $p$ to another process, the timestamp of the latest message delivered to $u_p$ from $p$}.

Example 4.1 and Figure 7 explain the behavior of the naive algorithm.

**Example 4.1** On receiving "*will-send$_q^r$(m)*" and "*will-send$_q^s$(m')*" successively from a user process $u_q$, process $q$ sends timestamped signals "$T_1$ : *request*" to $r$ and "$T_2$ : *request*" to $s$ ($T_1 < T_2$), and stores "$T_1$ : $r, m$, *requesting*" and "$T_2$ : $s, m'$, *requesting*" in $PQ_q$ in the order of increasing timestamps.

[5]Phases here are counted according to the number of signal transmissions.

1. **Request:** On receiving "*will-send$_p^q$(m)*" from a user process $u_p$, process $p$ sends a timestamped signal (not message $m$) "$T_p$ : *request*" to process $q$, where $T_p = (C_p, p)$ and $C_p$ is the current value of its clock, and stores the message $m$ marked as *requesting*, "$T_p$ : $q, m$, *requesting*", in its pending queue $PQ_p$.

2. **Permission:** On receiving a signal "$T$ : *request*" from $q$,

   - **if** ($T > LT_p$) **then** process $p$ stores "$T$ : *reserved*" in $PQ_p$, and sends a permission signal "$T$ : *permitted*($T$)" to $q$.
   - **if** ($T < LT_p$) **then** process $p$ stores "$T'$ : *reserved*" in $PQ_p$, where $T' = (C_p, p)$ and $C_p$ is the current clock of $p$, and sends "$T'$ : *permitted*($T$)" to $q$.

3. **Reception of Permission :** On receiving a signal "$T'$ : *permitted*($T$)" from $q$, process $p$ changes the stored message "$T$ : $q, m$, *requesting*" to "$T'$ : $q, m$, *permitted*" in $PQ_p$, and reorders $PQ_p$ in the order of increasing timestamps.

4. **Message send:** If the head of $PQ_p$ has a permitted message "$T'$ : $q, m$, *permitted*", then process $p$ sends a message "$T'$ : $m$" to process $q$, sets $LT_p = T'$, removes "$T'$ : $q, m$, *permitted*" from $PQ_p$, and notifies user process $u_p$ of the occurrence of the send event. This is repeated as long as the head of $PQ_p$ is a permitted message.

5. **Message reception:** On receiving a message "$T'$ : $m$", process $p$ changes "$T'$ : *reserved*" to "$T'$ : $m$" in $PQ_p$. On receiving a null message "$T'$ : *null*" (to be mentioned in Step 6), process $p$ just removes "$T'$ : *reserved*" from $PQ_p$.

6. **Message delivery:** If there is a message "$T'$ : $m$" at the head of $PQ_p$, then $p$ delivers $m$ to its user process $u_p$, sets $LT_p = T'$, and removes "$T'$ : $m$" from $PQ_p$. Upon delivery, the user process changes its state, and might decide to withdraw a message $m$ whose "*will-send(m)*" has been issued but has not yet been sent out. If user process $u_p$ changes only the message content from $m$ to $m'$ (and does not change its destination), then $u_p$ changes "$T$ : $q, m, *$" (* means *requesting* or *permitted*) to "$T$ : $q, m', *$". If $u_p$ cancels its issued "*will-send(m)*", then $u_p$ changes "$T$ : $q, m, *$" to "$T$ : $q, null, *$". This is repeated as long as the head of $PQ_p$ is a received message.

Figure 5: Naive algorithm $A_1$ for control process $p$

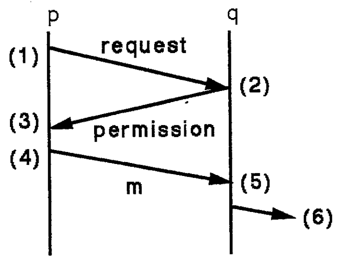


Figure 6: Naive algorithm $A_1$

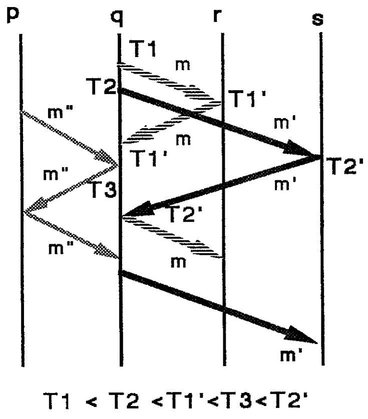


Figure 7: Behavior example of $A_1$

Assume that $LT_r > T_1$ and $LT_s > T_2$. On receiving a signal "$T_1$ : *request*" from $q$, process $r$ stores "$T_1'$ : *reserved*" in $PQ_r$ and sends "$T_1'$ : *permitted*($T_1$)" to $q$, where $T_1' = (C_r, r)$ and $C_r$ is the current clock value of $r$. Similarly, on receiving a signal "$T_2$ : *request*" from $q$, process $s$ stores "$T_2'$ : *reserved*" in $PQ_s$ and sends "$T_2'$ : *permitted*($T_2$)" to $q$, where $T_2' = (C_s, s)$ and $C_s$ is the current clock value of $s$. Here, we assume that $T_2 < T_1' < T_2'$.

On receiving "$T_1'$ : *permitted*($T_1$)" from $r$, $q$ changes "$T_1$ : $r, m$, *requesting*" in $PQ_q$ to "$T_1'$ : $r, m$, *permitted*" and reorders $PQ_q$. Since $T_2 < T_1'$, $q$ cannot send $m$ to $r$ before "$T_2$ : $s, m'$, *requesting*" is removed.

Before receiving "$T_2'$ : *permitted*($T_2$)", let us assume that $q$ transmits "$T_3$ : *permitted*($T_3$)" to $p$ and stores "$T_3$ : *reserved*" in $PQ_q$, where $T_2 < T_1' < T_3 < T_2'$. On receiving "$T_2'$ : *permitted*($T_2$)" from $s$, $q$ changes "$T_2$ : $r, m'$, *requesting*" in $PQ_q$ to "$T_2'$ : $r, m'$, *permitted*", and reorders $PQ_q$. Then, since the head of $PQ_q$ is "$T_1'$ : $r, m$, *permitted*", $q$ immediately sends a message "$T_1'$ : $m$" to $r$ and removes the corresponding message "$T_1'$ : $r, m$,*permitted*" from $PQ_q$. However, $q$ must wait at this point because "$T_3$ : *reserved*" is at the head of $PQ_q$. On receiving "$T_3$ : $m''$" from $p$ , $q$ changes "$T_3$ : *reserved*" to "$T_3$ : $m''$" in $PQ_q$, and $q$ delivers it to $u_q$, removing it from $PQ_q$. Let us assume that $q$ still wants to send $m'$ to $s$. Since "$T_2'$ : $s, m'$,*permitted*" is now at the head of $PQ_q$, $q$ sends message "$T_2'$ : $m'$" to $s$.

□

## 4.2 Correctness proof

First, we will prove that this algorithm has no-message-crossing property NMC.

**Theorem 4.1** *The above algorithm $A_1$ achieves NMC; i.e., for any $H \in S(NMC)$, there is no pair of messages $m$ and $m'$ in $H$ such that $(m \prec m') \wedge (m' \prec m)$.*

**Proof:** Similar to the proof of the third predicate in Theorem 3.1, let us assume that there is a pair of messages $m$ and $m'$ in $H$ such that $(m \prec m') \wedge (m' \prec m)$; i.e., there is a sequence of messages $MS : m = m_0, m_1, m_2, \ldots, m_{k-1} = m'$ such that $m_i \prec_{p_i} m_{i+1(mod\ k)}$ for $i \in \{0, 1, 2, \ldots, k-1\}$.

Case 1: Every relation $m_i \prec_{p_i} m_{i+1}$ in $MS$ is the $(+,-)$ relation.

This can be proven in the same way as in Case 1 of the third predicate in Theorem 3.1.

Case 2: Every relation $m_i \prec_{p_i} m_{i+1}$ in $MS$ satisfies the $(-,+)$ relation (see Fig. 3 (a)).

Let the timestamps of a message $m_i$ $(i = 0, \ldots, k-1)$ be $T_i$. When the send event of $m_i$ occurs in process $p_i$ before the delivery event of $m_{i+1}$, if time $T_{i+1}$ has been reserved in $PQ_{p_i}$ for $m_{i+1}$ before sending $m_i$, then $T_i < T_{i+1}$ because "$T_i$ : $p_{i-1}, m_i$, *permitted*" is the head of $PQ_{p_i}$; otherwise, $m_{i+1}$ will be permitted with $T_{i+1}$ which is larger than $LT_i \geq T_i$. Thus, we get $T_0 < T_1 < T_2 < \ldots < T_{k-1} < T_0$; this is a contradiction.

Case 3: $MS$ has the $(+,+)$ relation $+m_i \prec_{p_i} +m_{i+1}$ (see Fig. 3(b)).

Similar to the proof of Case 3 of the third predicate in Theorem 3.1, let $m_j \prec_{p_j} m_{j+1}$ be the first $(-,-)$ relation found by tracing $MS$ from $m_{i+1}$, and $m_k \prec_{p_k} m_{k+1}$ be the first $(+,+)$ relation found by further tracing $MS$ from $m_{j+1}$.

Let the timestamps of message $m_i$ $(i = 0, \ldots, k-1)$ be $T_i$. From the same reason as in Case 2, we get $T_{i+1} < T_{i+2} < \ldots < T_j$. Since the message at the head of $PQ_{p_j}$ is sent by $p_j$, and $p_j$ requests sending with the timestamp of its current clock value, we get $T_j < T_{j+1}$. When the delivery event of $m_{j+1}$ occurs in process $p_{j+1}$ before the send event of $m_{j+2}$, if $p_{j+1}$ has sent the request of sending $m_{j+2}$ before delivering $m_{j+1}$, then $T_{j+1} < T_{j+2}$ because "$T_{j+1}$ : $m_{j+1}$" is the head of $PQ_{p_{j+1}}$; otherwise, $T_{j+2} > T_{j+1}$ because $m_{j+2}$ will be requested with a timestamp of the current clock value of $p_{j+1}$, which is larger than $T_{j+1}$ and not larger than $T_{j+2}$. Thus, we get $T_{i+1} < T_{i+2} < \ldots < T_j < T_{j+1} < \ldots < T_k < T_{k+1}$. Tracing further from $m_{k+1}$ and using similar discussions, we can get $T_{k+1} < \ldots < T_i < T_{i+1}$; this is a contradiction.

□

Next, we will prove that this algorithm is free from deadlock.

**Theorem 4.2** *The naive algorithm $A_1$ is free from deadlock.*

**Proof:** Process $p_0$ must refrain from sending a permitted message "$T_0$ : $m_0$" to $p_k$, if there is either a requesting message "$T_1$ : $q, m'$, *requesting*", another permitted message "$T_1$ : $q, m'$, *permitted*", or a reserved message "$T_1$ : *reserved*" at the head of $PQ_{p_0}$. Since the requesting message at the head of $PQ_{p_0}$ will never fail to become a permitted message and the permitted message at the head will never fail to be sent and removed from $PQ_{p_0}$, it is sufficient to consider the case that a reserved message "$T_1$ : *reserved*"exists at the head of $PQ_{p_0}$. If $p_0$ has issued a permission signal "$T_1$ : *permitted*($\cdots$)" to $p_1$ and has stored "$T_1$ : *reserved*" in $PQ_{p_0}$, and if $T_1 < T_0$, then $p_0$ must refrain from sending a permitted message "$T_0 : m_0$" to $p_k$ until it receives a timestamped message "$T_1 : m_1$" from $p_1$. By a similar discussion, deadlock might occur when $p_1$ refrains from sending $m_1$ until receiving "$T_2 : m_2$" ($T_2 < T_1$), $p_2$ refrains from sending $m_2$ until receiving "$T_3 : m_3$" ($T_3 < T_2$), ... , $p_k$ refrains from sending $m_k$ until receiving "$T_0 : m_0$" ($T_0 < T_k$). This case however means that $T_0 > T_1 > T_2 > \ldots > T_k > T_0$: a contradiction. □

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

**Theorem 4.3** *The naive algorithm $A_1$ satisfies the fairness condition: 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.*

**Proof:** Any sender $p$ can send a request signal at any time and the receiver $q$ will return the corresponding permission signal with timestamp $T'$ immediately after receiving the request signal. Since the number of processes in a system is finite and the timestamps of the successive requests from a process are always incremented, the number of messages requested or permitted from $p$ with timestamp $T < T'$ is finite. From this and the freedom from deadlock, the sender $p$ will eventually be able to send the message to $q$.

□

## 4.3 Complexity

### 4.3.1 Complexity indices

We consider here the following *message complexity* and *time complexity* of an algorithm $A$.

- **Message Complexity $CM(A)$:** the maximum number of signals required to send a message if the sender continually wishes to send the message to the receiver.
- **Time Complexity $CT(A)$:** the maximum delay from receiving a "*will-send*" request to the occurrence of the corresponding send event, where an upper bound on interprocess communication delay is given as $T$, and the processing time of each process and the unit time of the clock can be considered to be negligible compared with $T$.

### 4.3.2 Lower bounds of complexity indices

First, we will present a trivial lower bound on $CM$ for any algorithm achieving NMC. Note that obtaining the strict lower bounds on $CM$ remains for further study.

**Theorem 4.4** *For any algorithm A achieving NMC, $CM(A) \geq 2$.*

**Proof:** Assume that there is an algorithm $A'$ achieving NMC by sending only one signal for each message. At the initial state of the algorithm $A'$ achieving NMC, only one process $p$ must obtain a permission to send a message. In order to achieve NMC, the other processes have to receive a signal before obtaining permission to send a message, because each process can get information only by exchanging signals. If there is a process that has received a signal but will not send a message, then the algorithm $A'$ cannot proceed; a contradiction. Thus, we get $CM(A) \geq 2$.

□

We will next consider lower bounds on $CT$ for algorithms achieving NMC. The message sequence $m_k \prec m_{k-1} \prec \ldots m_i \prec m_{i-1} \prec \ldots \prec m_1$ such that $m_i \prec m_{i-1}$ ($i = 2, \ldots, k$) has the $(-,+)$ relation is called a *dependent message sequence*; it is said that message $m_1$ *depends on* $m_i$ ($i = 2, \ldots, k$). On the other hand, the message sequence $m'_k \prec m'_{k-1} \prec \ldots m'_i \prec m'_{i-1} \prec \ldots \prec m'_1$ such that $m'_i \prec m'_{i-1}$ ($i = 2, \ldots, k$) has the $(+,-)$ relation is called a *message chain*.

**Theorem 4.5** *If an algorithm A achieving NMC does not use global information (i.e., the processes involved in determining whether a message transmission can be executed or not are only the corresponding sender and receiver processes), then $CT(A) \geq 2kT$, where k is the maximum length of the dependent message sequence.*

To prove this theorem, the process' *knowledge* concept must be clearly defined. We will follow the notation used by Halpern and Moses[7] to describe the knowledge of processes. We start with the primitive propositions in $\Phi$ and form more complicated predicates $\varphi$ by closing off under negation $\neg$, conjunction $\wedge$, and modal operators $K_1, \ldots, K_n$ (one for each process). A statement like $K_i\varphi$ is read "*process i knows* $\varphi$."

A process' knowledge at the end of a history is defined inductively on the predicate $\varphi$. The truth assignment $\pi$ tells us, for each history $H$ and each primitive proposition $\rho \in \Phi$, whether $\rho$ is true or false at the end of $H$. "$\varphi$ is true at the end of $H$," is written as $H \models \varphi$.

- $H \models \rho$ (for a primitive proposition $\rho \in \Phi$) iff $\pi(H, \rho) =$ **true**,
- $H \models \varphi \wedge \psi$ iff $H \models \varphi$ and $H \models \psi$,
- $H \models \neg\varphi$ iff $H \not\models \varphi$, and
- $H \models K_i\varphi$ iff $H' \models \varphi$ for all $H'$ such that $H$ and $H'$ are equivalent to $i$, i.e., $H \overset{i}{\sim} H'$.

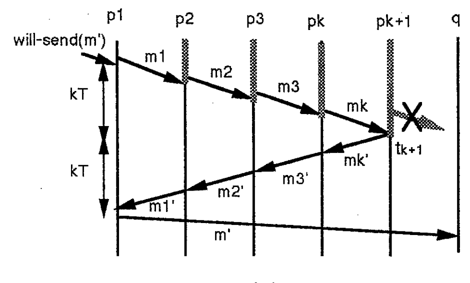
The last clause captures the intuition that process $i$ knows a predicate $\varphi$ in $H$ exactly, if $\varphi$ is true for all histories $i$ can consider possible for $H$.

Chandy and Misra[5] defined a predicate $\varphi$ to be *local* to a process $i$ if for all histories $H$, $H \models K_i\varphi \vee K_i\neg\varphi$. That is, the value of $\varphi$ is *always* known to $i$. They showed the following lemma.

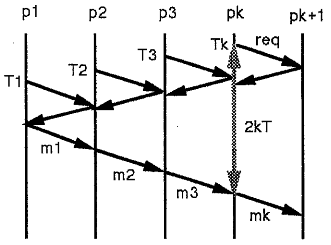
**Lemma 4.1** *([5]) Let $p_1, p_2, \ldots, p_{k+1}$ be processes in a system, H and H′ be histories such that H is a prefix of H′, and $\varphi$ be a predicate that is local to $p_{k+1}$. If $H \models \neg(K_{p_k}\varphi)$ and $H' \models K_{p_1}(K_{p_2}(\ldots(K_{p_k}(\varphi))))$, then there is a message chain $m'_k \prec m'_{k-1} \prec \ldots \prec m'_2 \prec m'_1$ from $p_{k+1}$ to $p_1$, which is contained in H′ but not in H.*

**Proof of Theorem 4.5:**
Consider the case in which a process $p_1$ has sent $m_1$ to $p_2$ and immediately after that the user process $u_{p_1}$ issues "*will-send*$^q_{p_1}(m')$", and there is a dependent message sequence of length $k$, $m_k \prec m_{k-1} \prec \ldots, \prec m_1$, as shown in Fig. 8(a). It takes at worst $kT$ from sending $m_1$ to receiving $m_k$. Since $m'$ has a possibility to be instantaneously received, sending $m'$ to $q$ is permitted only if $p_1$ has the *knowledge* that there is no undelivered message to the user process of $q$, $u_q$, on which $m_1$ depends.

If global information can be used, then $p_1$ can get the above knowledge in $KT + T$ time after sending $m_1$, by letting each process broadcast all the necessary information to other processes on receiving a message. However, without global information, $p_1$ requires the knowledge $K_{p_1}$(there is no undelivered message to $u_q$ on which $m_1$ depends). For a dependent message sequence of length $k$ (note that the maximum length $k$ is assumed to be unknown to each process) the above knowledge



(a)



(T1 < T2 < T3 < ... < Tk)

(b)

Figure 8: Explanation for lower bound of time complexity

is equal to $K_{p_1}(K_{p_2}$(there is no undelivered message to $u_q$ on which $m_2$ depends)), where $m_2$ is the message sent by $p_2$ before delivering $m_1$. Similarly, the above knowledge is equal to $K_{p_1}(K_{p_2}(K_{p_3} \ldots (K_{p_{k+1}}$(there is no undelivered message to $u_q$ at $t_{k+1}$)))), where $t_{k+1}$ is the time instant that $m_k$ is delivered to $u_{p_{k+1}}$.

Since the predicate "there is no undelivered message to $u_q$ at $t_{k+1}$" is *local* to $p_{k+1}$, but not local to the other processes, there is by Lemma 4.1 a message chain $m'_k \prec \ldots \prec m'_2 \prec m'_1$ transmitted after delivering $m_k$ but before sending $m'$; thus we get $CT \geq 2kT$. □

### 4.3.3 Complexities of the naive algorithm

Since the naive algorithm $A_1$ always requires three signals, "$T$ : *request*", "$T'$ : *permitted*($T$)", and "$T'$ : $m$" to send a message $m$, the message complexity of algorithm $A_1$, $CM(A_1)$ is equal to 3. The time complexity of $A_1$, $CT(A_1)$, is equal to $2kT$, where $k$ is the maximum length of the dependent message sequence. This is because sender process $p_k$ must wait at most 2T time from receiving "*will-send*$_{p_k}^{p_{k+1}}(m)$" to receiving the permission "$T_k$ : *permitted*($T_k$)" from $p_{k+1}$, and must wait for receiving a message "$T_{k-1}$ : $m_{k-1}$" before sending the message "$T_k$ : $m_k$" if $p_k$ has returned a permission signal "$T_{k-1}$ : *permitted*($T_{k-1}$)" to $p_{k-2}$ and $T_{k-1} < T_k$. As shown in Fig. 8(b), the waiting time from receiving "$T_k$ : *permitted*($T_k$)" to sending "$T_k$ : $m_k$" is at most $2(k-1)T$ if the maximum length of the dependent message sequence is equal to $k$. Thus, from these and the lower bounds in Theorem 4.4 and Theorem 4.5, $CT(A_1)$ is optimum and $CM(A_1)$ is quasioptimum (at most one larger than the lower bound).

As references, the complexities of Goldman's algorithm $G$ are: $CM(G) = 4$ and $CT(G) = 4(k+1)T$[6], and those of Bagrodia's rendezvous algorithm $RDV$ are: $CM(RDV) = 4n - 3$ and $CT(RDV) = 4(n-1)T$, where $n$ is the number of processes

1. **Request:** Same as Step 1 of algorithm $A_1$ except for setting $MT_p = T_p$.
2. **Permission:** On receiving a signal "$T$ : *request*" from $q$,
   - **if** ($T > MT_p$) **then** process $p$ stores "$T$ : *reserved*" in $PQ_p$, and sends "$T$ : *permitted*($T$)" to $q$.
   - **if** ($LT_p < T < MT_p$) **then** process $p$ chooses $T_p$ as either $T$ or $T'$ at random, stores "$T_p$ : *reserved*" in $PQ_p$, and sends "$T_p$ : *permitted*($T$)" to $q$, where $T' = (C_p, p)$ and $C_p$ is the current clock of $p$.
   - **if** ($T < LT_p$) **then** process $p$ stores "$T'$ : *reserved*" in $PQ_p$ and sends "$T'$ : *permitted*($T$)" to $q$.
4. **Message send, 6. Message delivery:** Same as Steps 4 and 6 of algorithm $A_1$ except for setting $MT_p = T'$ and $LT_p = T'$ instead of $LT_p = T'$.

Figure 9: Improved algorithm $A_2$ for control process $p$

in a system[1]. Namely, the naive algorithm $A_1$ is better than Goldman's algorithm and Bagrodia's algorithm in $CM$ and $CT$. Although RPC is efficient with respect to its complexities, i.e., $CM(RPC) = 2$ and $CT(RPC) = 2kT$[3], it may cause deadlocks in the partner model.

## 5 Improvement of average-case behaviors by randomization

This section proposes another algorithm $A_2$ in order to improve the average-case time complexity of the naive algorithm $A_1$ by using a randomization technique. Let us consider the case in Fig. 8(b) which gives the worst delay, i.e., $CT(A_1) = 2kT$. Let each process $p_i$ request with timestamp $T_i$ ($T_1 < T_2 < \ldots < T_k$) and $T'_i$ be the timestamp that $p_i$ is permitted to send. If $T_i = T'_i$ for every $i$, this worst case occurs. However, if $T'_1 < T'_2$, $T'_2 > T'_3$, $T'_3 < T'_4, \ldots$, then the *message chain* $m_1 \prec m_2 \prec \ldots \prec m_k$ is cut separately and the worst delay in this case becomes shorter. It can be easily shown that algorithm $A_1$ still works even if the timestamp $T'_q$ that each process $p$ gets from $q$ is set to any value not less than $\max\{T_p, LT_q + 1\}$, where $LT_q$ is identical to the one defined in the algorithm $A_1$. Therefore, the sending pattern can be changed by chosing the value of $T'_p$ appropriately.

The algorithm $A_2$ given in Fig. 9 has an improved average delay by technique that randomly selects $T'_q$. Here, $MT_p$ is max {the timestamp of the latest message sent or requested from $p$ to another process, the timestamp of the latest message delivered to $u_p$ from $p$}, and $LT_p$ is as defined for algorithm $A_1$. All the other steps are the same as those for $A_1$.

Here, we will evaluate the average of all sending patterns, as shown in Fig. 8(b), for $k = 4$. For simplicity, we assume that $LT_{i+1} < T_i$ ($i = 1, \ldots, 4$). For example, consider $m_1$ and $m_2$ in Fig. 8(b), then the permitted timestamp for sending $m_1$ will be $T_1$ or $T'_2 (> T_2)$ and that for sending $m_2$ will be $T_2$ or $T'_3 (> T_3)$. When the permitted timestamp of $m_1$ is $T'_2$ and that of $m_2$ is $T_2$, $p_2$ can send $m_2$ without waiting for the reception of $m_1$ because $T_2 < T'_2$.

Let $(r_1, r_2, r_3, r_4)$ be a tuple of the permitted timestamps of $m_1$, $m_2$, $m_3$, and $m_4$. We will evaluate the worst delay $WD$ for all sending patterns and their average $CT_{av}$.

$WD$ and $CT_{av}$ for $A_2$

| WD | $(r_1, r_2, r_3, r_4)$ | Probability |
|---|---|---|
| $2kT = 8T$ | $(T_1, T_2, T_3, T_4)$ | $\frac{1}{8}$ |
| $2(k-1)T = 6T$ | $(T_1, T_2, T'_4, T_4)$, $(T_1, T'_3, T'_4, T_4)$ $(T'_2, T'_3, T'_4, T_4)$, $(T'_2, T_2, T_3, T_4)$ | $\frac{1}{2}$ |
| $2(k-2)T = 4T$ | $(T_1, T'_3, T_3, T_4)$, $(T'_2, T'_3, T_3, T_4)$ $(T'_2, T_2, T'_4, T_4)$ | $\frac{3}{8}$ |
| $CT_{av}$ | $5.5T$ | |

The average of WD, $CT_{av}(A_2) = 5.5T$, is an improvement from $CT(A_1) = 8T$.

Note that this randomization does not increase the delay $WD$ even for the other cases by the following reason. Consider the case of $k = 3$. If $T_1 < T_2$ and $T_3 < T_2$, then the randomization might cut the message chain $m_1 \prec m_2$, but does not influence the order of $m_2$ and $m_3$. If $T_1 > T_2 > T_3$, then the randomization does not work for this case. If $T_1 > T_2$ and $T_3 > T_2$, then the permitted timestamp of $m_2$ is chosen randomly as $T_2$ or $T'_2(> T_3)$. If $T_2$ is chosen, then the message sending pattern is the same as that of naive algorithm $A_1$. If $T'_2$ is chosen, then the randomization creates a new message chain $m_1 \prec m_2$ but cuts the original message chain $m_2 \prec m_3$ separately, and thus does not increase $WD$. Thus, the randomization does not increase $WD$ in every case.

## 6 Conclusions

This paper presents a mechanism *simulated* by instantaneous message passing in asynchronous systems. A necessary and sufficient condition is first clarified for simulated by instantaneous message passing, which enables us to devise more efficient algorithms than the previously proposed ones: Bagrodia's rendezvous algorithm[1] and Goldman's algorithm[6]. A naive algorithm is next proposed for this mechanism whose worst-case message complexity is quasioptimum and time complexity is optimum. Furthermore, a modified algorithm is proposed for attaining higher efficiency in the average case evaluation by using a randomization technique.

Extension to multicast communications remains for further study, which might improve the result of Goldman[6].

**Acknowledgements**

The first author would like to thank Sam Toueg of Cornell University for his discussions on the earlier stages of this work, and also Yoshifumi Manabe and Haruhisa Ichikawa for their comments on earlier drafts of this paper.

## 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] A. D. Birrel and B. J. Nelson, "Implementing Remote Procedure Calls,"ACM Trans. on Computer Systems, Vol.2, No.1, pp.39-59 (1984).

[4] G. V. Bochmann, "Specifications of a simplified transport protocol using different formal description techniques," Computer Networks and ISDN Systems, Vol. 18, pp.335-377 (1989/1990).

[5] K. M. Chandy and J. Misra, "How processes learn," Distributed Comput. Vol.1, pp. 40-52 (1986).

[6] K.J. Goldman, "Highly concurrent logically synchronous multicast," Lecture Notes in Computer Science, Vol. 392, *Distributed Algorithms*, pp.94-109 (1989).

[7] J. Y. Halpern and Y. Moses, "Knowledge and common knowledge in a distributed environment," Journal of the ACM, Vol.37, No.3, pp.549-587 (1990).

[8] S. Harashima and T. Ibaraki, "Concurrency control of distributed database systems by cautious schedulers," Trans. of IEICE Japan, Vol. J70-D, No. 6, pp.1140-1148 (in Japanese) (1987).

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

[10] 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).

[11] G. Neiger and S. Toueg, "Substituting for real time and common knowledge in asynchronous distributed systems," Proc. of 6th ACM Symp. on Principles of Distributed Computing, pp.281-293 (1987).

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

[13] T. Soneoka, "Instantaneous message passing in asynchronous distributed systems," IEICE Technical Report, COMP 90-70, pp.79-86 (1990).