Synchronous Message Passing

V. V. Murty and V. K. Garg

TR ECE-PDS-93-01 October 1993

Parallel & Distributed Systems group

Department of Electrical & Computer Engineering

University of Texas at Austin

Austin, Texas 78712

Synchronous Message Passing

V. V. Murty and V. K. Garg
Department of Electrical & Computer Engineering
University of Texas at Austin
email: murty, vijay@pine.ece.utexas.edu

Abstract

This paper studies the characteristics of synchronous ordering of messages. Synchronous ordering of messages defines synchronous communication based on the causality relation rather than time. We present the necessary characteristics of any algorithm providing deadlock-free synchronous ordering of the messages. We also present the sufficient conditions, based on the causality relations, for any algorithm to provide synchronous ordering. The paper proposes an algorithm using acknowledgment messages to implement the sufficient conditions. The acknowledgment messages are used to satisfy the causality relation between the events. The algorithm is deadlock-free, and provides a higher degree of concurrency then the algorithms which define synchronous communication based on time.

1 Introduction

Distributed programs are difficult to design and test due to their non-deterministic nature. That is, a distributed program may exhibit multiple behaviors on the same external input. This non-determinism is caused by a possible reordering of messages in different executions. It is sometimes desirable to control the non-determinism by restricting the possible message ordering in a system. For example, many systems restrict message delivery to FIFO order. This results in simplicity in design of distributed algorithms which may depend on the FIFO assumption [3]. In this paper, we discuss one such possible restriction on message ordering called synchronous message ordering.

The problem of restricting message ordering has received wide attention [1, 7, 12, 13, 10]. Charron-Bost, Mattern, and Tel [1] have recently proposed a strict hierarchy of message ordering - asynchronous, FIFO, causal ordering, and synchronous.

An asynchronous computation does not have any restriction on the message ordering. It is easy to implement, and it permits maximum concurrency; but algorithms based on fully asynchronous communication can be difficult to design. These algorithms are required to work for all ordering of the messages. A FIFO ordered computation is also easy to implement. It is usually implemented by using sequence numbers for messages. Causal ordering is a stronger condition than FIFO. Intuitively, causal ordering requires that a single message should not be overtaken by a sequence of messages. Joseph and Birman [7] have given many examples of problems which are easier to solve if causal ordering is assumed. It was first implemented in ISIS [4]. Many other algorithms for causal ordering have appeared since then [12, 13]. Synchronous ordering is a stronger requirement than causal ordering. Algorithms for synchronous systems are much easier to design than those for causally ordered systems. Unlike the case of causal ordering, there is very little work done on synchronous ordering of messages. Usually, to achieve synchronous ordering of messages, the send is blocking, i.e., the sender waits for an acknowledgment from the receiver to execute the next event. This informal algorithm as presented by Charron-Bost, Mattern, and Tel [1] can result in deadlocks.

In this paper, we present a deadlock free algorithm that ensures synchronous ordering of messages. In addition, our algorithm permits higher degree of concurrency than algorithms based on blocking sends. In our algorithm the sender continues to execute internal events and receive events while waiting for an acknowledgment message.

This paper is organized as follows. In Section 2, we give formal definitions of various message orderings in distributed computations. This section also includes background results on characterization of synchronous computation using crowns [1]. In Section 3, we investigate the requirements of any algorithm that achieves synchronous ordering. Note that message ordering can be restricted, in general, by delaying sends or by delaying receives. For example, the causal ordering algorithm proposed by Raynal, Schiper, and Toueg [12] uses receiver based delays. We show in this section that any algorithm that implements synchronous ordering must include both sender and receiver based delays. In Section 4, we present our algorithm and Section 5 provides the proof of its correctness. Section 6 discusses the overhead associated with the algorithm and presents concluding remarks.

2 Distributed Computations

2.1 Our Model

Let a distributed system be defined as a set of n sequential processes $\{P_i \,|0 \leq i < n\}$. Each process P_i consists of a set of events $\mathcal{E}$, which are classified into three types: *send* events $\mathcal{S}$, *receive* events $\mathcal{R}$, and *internal* events $\mathcal{I}$. The next event that is executed by process P_i after an event e is given by the relation $\prec_1$. The events in the same process form a total order under the transitive closure of $\prec_1$.

We assume that every *send* event has a corresponding *receive* event, i.e., every message that is sent is eventually received. We also assume that every *receive* event has a corresponding *send* event, i.e., there are no spurious messages in the system. A send event s in process P_i and its corresponding receive event r in process P_j $(j \neq i)$ are related by the the relation $\rightsquigarrow$. This is represented as $s \rightsquigarrow r$. For convenience, we denote the receive event corresponding to the send event s_i by r_i and vice–versa. The message pair is represented as (s_i, r_i)). Thus, $\forall i \,:\, s_i \rightsquigarrow r_i$.

The causal ordering of events in distributed systems is based on the well known "happened before $(\rightarrow)$" relation [8]. The happened before relation (also called causally precedes relation) is defined as the transitive closure of union of $\prec_1$ and $\rightsquigarrow$ relations. In other words, $e \rightarrow f$ iff

1. $(e \prec_1 f) \,\vee\, (e \rightsquigarrow f)$, or
2. $\exists h \,:\, (e \rightarrow h \,\wedge\, h \rightarrow f)$.

Similarly, the relation $\prec$ is defined as the transitive closure of $\prec_1$, that is, $e \prec f$ iff

1. $e \prec_1 f$, or
2. $\exists h \,:\, (e \prec h \,\wedge\, h \prec f)$.

For example in Figure 1, the events a, b, c, e, f are related as $a \prec_1 b$, $b \prec_1 c$, $b \rightsquigarrow e$ and $e \prec_1 f$; therefore, the events a and f are related as $a \rightarrow f$, and the events a and c are related as $a \prec c$ (therefore, $a \rightarrow c$).

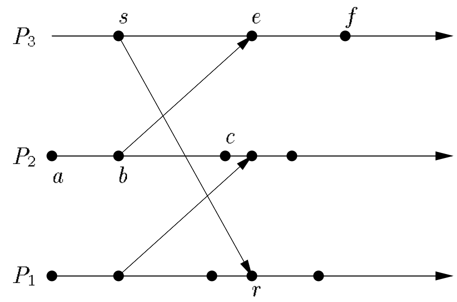


Figure 1: A Time Diagram for a Distributed run

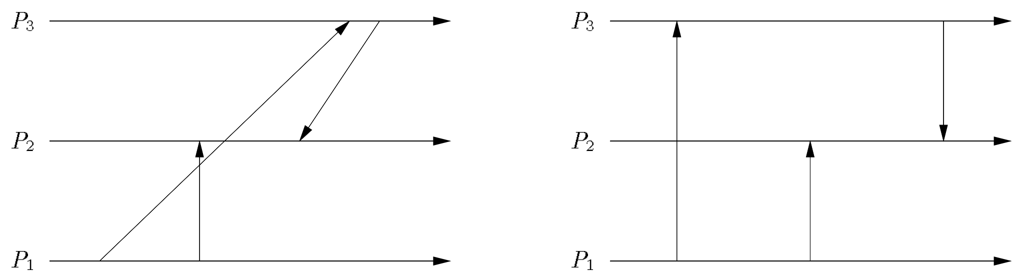


Figure 2: Synchronous Time-diagram

2.2 Hierarchy of Communication Modes

Based on the happened before relation a strict hierarchy of communication modes can be defined [1]. The various communication modes are:

FIFO : Any two messages from a process P_i to P_j are received in the same order as they were sent. Formally,

$$s_1 \prec s_2 \implies \neg(r_2 \prec r_1). \qquad \text{(FIFO)}$$

Causally Ordered : Let any two send events s_1 and s_2 in a distributed computation be related such that the first send causally precedes the second send. Then, the second message cannot be received before the first message by any process [7, 12]. Formally,

$$s_1 \rightarrow s_2 \implies \neg(r_2 \prec r_1). \qquad \text{(CO)}$$

Synchronous : A computation is synchronous if its time diagram can be drawn such that all message arrows are vertical [1] (see Figure 2). That is, all external events can be assigned a timestamp such that time increases within a single process and for any message its send and receive are assigned the same timestamp. Formally,

$$\begin{aligned} \exists \mathrm{T} : \mathcal{E} \longrightarrow \mathbb{N} \quad &: \quad \forall s, r, e, f \in \mathcal{E} \setminus \mathcal{I} \\ s \leadsto r \quad &\implies \quad \mathrm{T}(s) = \mathrm{T}(r) \qquad \text{(SYNC)} \\ e \prec f \quad &\implies \quad \mathrm{T}(e) < \mathrm{T}(f). \end{aligned}$$

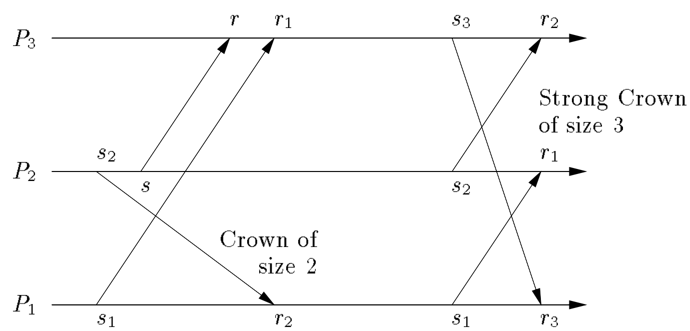


Figure 3: Crowns of size 2 and 3

This definition is different from that given in [1] but it can easily be shown to be equivalent. In the rest of the paper, we assume that there are no internal events, as they do not affect the message ordering; therefore, $\mathcal{E} = \mathcal{R} \cup \mathcal{S}$. It is easy to see that, for any two events e and f

$$(e \rightarrow f) \wedge \neg(e \rightsquigarrow f) \quad \Longrightarrow \quad T(e) < \mathrm{T}(f). \tag{1}$$

The following theorem [1] is proved for the sake of completeness.

Theorem 1 *The hierarchy associated with the various modes of communications is*

$$\textit{Synchronous} \subseteq \textit{Causally Ordered} \subseteq \textit{FIFO}.$$

Proof:

Causally Ordered $\subseteq$ FIFO : This is true because,

$$s_1 \prec s_2 \quad \Longrightarrow \quad s_1 \rightarrow s_2.$$

Synchronous $\subseteq$ Causally Ordered : We show that if a computation is synchronous then it is also causally ordered. Since the communication is synchronous there exists a function T satisfying SYNC.

For any set of events $s_1, s_2 \in \mathcal{S}$ and $r_1, r_2 \in \mathcal{R}$ such that $s_1 \rightsquigarrow r_1$, $s_2 \rightsquigarrow r_2$ and $s_1 \rightarrow s_2$:

$$\mathrm{T}(s_1) = \mathrm{T}(r_1) \quad \mathrm{T}(s_2) = \mathrm{T}(r_2), \quad \text{and} \quad \mathrm{T}(s_1) < \mathrm{T}(s_2).$$

It follows that $\mathrm{T}(r_1) < \mathrm{T}(r_2)$. Therefore, (1) implies

$$\neg(r_2 \rightarrow r_1).$$

||

2.3 Crowns in a Distributed Computation

A computation can also be characterized as synchronous based on absence of a structure called "crown". The concept of a crown was introduced by Charron-Bost et al., in [1] where they also prove that a computation is synchronous if and only if it does not contain any crown. In the rest of the section we present the definition of a crown and provide a simpler proof of this property.

Definition (Crown): *Let C be a computation. A* crown *(of size k) is a sequence $\langle (s_i, r_i), i \in \{0, 1 \ldots, k-1\} : s_i \rightsquigarrow r_i \rangle$ of pairs of corresponding send and receive events such that (see Figure 3)*

$$s_0 \rightarrow r_1, s_1 \rightarrow r_2, \cdots, s_{k-2} \rightarrow r_{k-1}, s_{k-1} \rightarrow r_0.$$

Theorem 2 *A computation is synchronous iff there is no crown in it.*

Proof:

Synchronous $\Longrightarrow$ ¬Crown :

Since the computation is synchronous there exists a function T satisfying SYNC, and for any two events e and f

$$(e \rightarrow f) \wedge \neg(e \rightsquigarrow f) \quad \Longrightarrow \quad \mathrm{T}(e) < \mathrm{T}(f).$$

Suppose, if possible, the computation has a crown of size k,

$$s_0 \rightarrow r_1, s_1 \rightarrow r_2, \cdots, s_{k-2} \rightarrow r_{k-1}, s_{k-1} \rightarrow r_0.$$

Therefore,

$$\begin{array}{lll} \forall i \in \{0, 1, \ldots, k-1\} & \mathrm{T}(s_i) < \mathrm{T}(r_{(i+1) \bmod k}) & (*) \\ \forall i \in \{1, 2, \ldots, k-1\} & \mathrm{T}(s_i) = \mathrm{T}(r_i). & (**) \end{array}$$

Therefore, from equations (*) and (**),

$$T(s_0) < T(r_0).$$

which is a contradiction because SYNC implies that $T(s_0) = T(r_0)$.

¬Crown $\Longrightarrow$ Synchronous :

Given a computation, we form a directed graph $G(E, V)$, as follows. The vertex set V consists of all messages in the computation. Thus, each vertex v_i represents a set of two events: the send event s_i and the corresponding receive event r_i. That is,

$$v_i = \{s_i, r_i\}.$$

There is an edge from v_i to v_j if there is an event $e \in v_i$ and an event $f \in v_j$ such that $e \rightarrow f$. Thus, $(v_i, v_j) \in E$ iff $(s_i \rightarrow s_j) \vee (s_i \rightarrow r_j) \vee (r_i \rightarrow s_j) \vee (r_i \rightarrow r_j)$. It is easy to see that each of the four disjuncts implies $s_i \rightarrow r_j$. Hence, $(v_i, v_j) \in E$ iff $s_i \rightarrow r_j$.

Since the computation does not have any crown, it follows that the graph G is acyclic.

This means that G can be topologically sorted [2]. Therefore, there exists a function F: $\mathcal{E} \rightarrow \mathbb{N}$ such that,

$$\begin{array}{lll} s \rightsquigarrow r & \Longrightarrow & \mathrm{F}(s) = \mathrm{F}(r) \quad \text{and} \\ e \prec f & \Longrightarrow & \mathrm{F}(e) < \mathrm{F}(f). \end{array}$$

Therefore, the computation is synchronous.

||

In this paper, we also use a structure called strong crown. We define a strong crown as,

Definition (Strong Crown): *Let C be a computation. A* strong crown *(of size k) is a sequence $\langle (s_i, r_i), i \in \{0, 1 \ldots, k-1\} : s_i \rightsquigarrow r_i \rangle$ of pairs of corresponding send and receive events such that*

$$s_0 \prec r_1, s_1 \prec r_2, \cdots, s_{k-2} \prec r_{k-1}, s_{k-1} \prec r_0.$$

Note that a strong crown is a crown, but not all crowns are strong (see Figure 3). Later to prove the safety of the algorithm we show that, any distributed computation satisfying the conditions of the algorithm, has a strong crown if it has a crown.

3 Some Impossibility Results

In this section we present the necessary charateristics of any algorithm that ensures synchronous ordering of the messages. The assumptions for any protocol are:

1. A protocol for any message is restricted to the processes executing the send event and the corresponding receive event. If there is a message (s, r) from process P_i to P_j, the protocol should not contain any control messages from P_k $(k \neq i, k \neq j)$ to P_i or P_j, or vice-versa.

2. A process can take a decision at $t = t_0$ about any event e (i.e., whether to delay it or to execute it) only based on the past. The past of any event e at time t is defined as:

$$\text{past}(e) = \{f | f \rightarrow e\}.$$

   The $\rightarrow$ includes the causality formed by the control messages.

A protocol consists of a receiver part and the sender part. A protocol is defined as *bounded delay send* if there exists an upper bound on the time to complete the sender part. Similarly, a protocol is *bounded delay receive* if there exists an upper bound on the time to complete the receiver part. Intuitively, a protocol is bounded delay receive if on receiving a message (s, r) at time $t = t_0$, the process commits (completes the protocol) the message by time $t = t_1 + \delta t$.

For example, consider FIFO ordering in a two process system. The usual protocol to implement FIFO is a *bounded delay send* but not a *bounded delay receive*. If a process intends to send a message, then it can immediately execute the send and end the sender part. Assume that there are two messages (s_1, r_1) and (s_2, r_2), and $s_1 \prec s_2$. If the receiver process receives r_2 at time t_0 before the message r_1, then it cannot assure the commit of the message in a bounded time. This unbounded time execution of the protocol is due to the uncertainty in the amount of time the message r_1 will take. In effect, the process is delaying the receive until some other event has taken place.

In this section we prove that any protocol that implement synchronous ordering must be asymmetric and must include sender and receiver based delays. We assume without loss of generality, that the upper bound on time is δt on the completion of the sender part or the receiver part, if there exist one. We also assume, without loss of generality, that to implement the protocol for the message (s_i, r_i), the first message of the protocol between two processes is represented as the message (s_i, r_i).

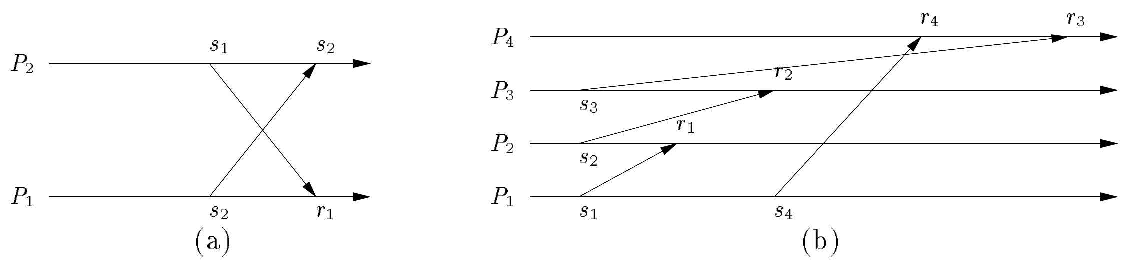


Figure 4: Non-synchronous ordering for impossibilities

Theorem 3 *Any protocol that implements SYNC cannot be*

1. *symmetric, or*
2. *have bounded delay sends, or*
3. *have bounded delay receives.*

Proof:

1. We show that given a symmetric protocol with respect to the processes, there exists a distributed computation where synchronous ordering is not possible.

   Consider the distributed computation shown in Figure 4(a). Both the processes intend to send a message to each other at the same time t_0. Since the past for each of the sends is an empty set, none of the processes can delay the send. It is obvious that the system is in a symmetric state with respect to P_1 and P_2. If this symmetric state is the input to any symmetric protocol, the resulting output cannot be asymmetric. This rules out the possibility of the asymmetric ordering of messages i.e., $(s_1 \prec r_2 \ \wedge \ r_1 \prec s_2)$ or $(s_2 \prec r_1 \ \wedge \ r_2 \prec s_1)$, which are the only possible synchronous message orderings.

   Therefore, either the resulting computation does not satisfy SYNC, or the processes will never send the message.

2. *Bounded delay receive* protocol:
   Consider the distributed computation as shown in Figure 4(b). Each of the processes P_1, P_2 and P_3 sends a message at time t_0. Since the past for each of the sends is an empty set, none of the processes can delay the send. The message (s_1, r_1) is received by process P_2 at time t_1. The process P_2 completes the protocol for message (s_1, r_1) by time $t_1 + \delta t$ since the protocol is *bounded delay receive*. At the completion of the protocol, process P_2 may not have any information about the event r_2. This is because the time taken for message (s_2, r_2) may have been greater than $t_1 + \delta t - t_0$.

   Consider message (s_1, r_1); the following are true:

   (a) Processes taking part in the protocol are P_1 and P_2.

   (b) Process P_2 completes the protocol at time $t_1 + \delta t$.

(c) Process P_2 cannot send any more messages to P_1 after time $t_1 + \delta t$, because of *Bounded delay receive.*

(d) When the process P_1 completes the protocol at say time t_2, it knows its past (i.e., $t \leq t_2$) and the past of the process P_2 before the completion of event r_1, i.e., $t \leq (t_1 + \delta t)$.

As a result of statement (d), the process P_1 has no knowledge of event r_2 at time t_2.

Now consider the message (s_4, r_4), which the process P_1 wants to execute at $t = t_3 > t_2$. From the assumption 2 of the protocol, the process P_1 executes the event s_4 based on its past, i.e.,

$$\text{past}(s_4) \subseteq \{s_1, s_2, r_1\}.$$

Based on the past, the process P_1 cannot delay the send of event s_4. Let the process P_4 receive the message (s_4, r_4) at time t_4. The process P_4 completes the protocol at time $t_4 + \delta t$ since the protocol is *bounded delay receive.*

If the message (s_3, r_3) takes more than $t_4 + \delta t - t_0$, then the process P_4 orders the receive events r_3, r_4 such that $r_4 \prec r_3$. The resulting distributed computation is non-synchronous as there is a crown.

3. *Bounded delay send* protocol:

Again, consider the distributed computation as shown in figure 4.(b), each process P_1, P_2, and P_3 send a message at time t_0.

Assume the process P_1 wants to send another message (s_4, r_4) at time t_1. Since the protocol is *bounded delay send*, the process P_1 executes s_4 and completes the send part of the protocol before $t = t_1 + \delta t$. If the message (s_1, r_1) takes more than $t_1 + \delta t - t_0$ units of time, then the message (s_4, r_4) carries no information about the event r_1. Therefore,

$$\text{past}(s_4) = \{s_1\}.$$

Let the message (s_4, r_4) reach the process P_4 at time $t = t_2$. On receiving the message r_4, the process has no knowledge of the message (s_3, r_3) since,

$$\text{past}(r_3) = \{s_1, s_4\}.$$

Therefore, the only possible action the process P_4 can take is to commit the receive of the message r_3 at time t_2. The message ordering results in a non-synchronous computation.

||

4 Algorithm

4.1 Commit Point of a Message

To implement synchronous ordering in the traditional algorithm, a process sends a message $s_1 \rightsquigarrow r_1$ and waits for an acknowledgment $s_2 \rightsquigarrow r_2$ before executing other events. Therefore, the message transaction is completed on the receive of the acknowledgment. In these systems, the

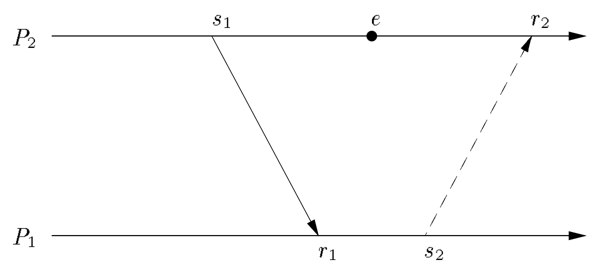


Figure 5: A message from P_2 to P_1

interval from s_1 to r_2 is atomic. That is, the sender does not execute any event between s_1 and r_2. In our algorithm, the send of a message and the receive of the corresponding acknowledgment result in completion of the message transaction, but the events s_1 and r_2 are not atomic. This gives a process flexibility to order the message as if it was sent at either s_1 or r_2. For example, in figure 5 the process starts the message transaction at event s_1 and ends the transaction at event r_2. If the process P_2 commits the message at s_1 the message causally precedes the event e, whereas if the process commits the message at r_2 the event e causally precedes the message.

4.2 Protocol

The algorithm to implement synchronous ordering of messages has three components: the priority rule (PR), the send condition (SC) and receive condition (RC). As shown in Section 3, any protocol that achieves synchronous ordering must be asymmetric with respect to processes. We introduce this asymmetry by the priority rule. The algorithm introduces control messages. These control messages are denoted by s^c and r^c and belong to the event set $\mathcal{E}^c$. Based on the event sets $\mathcal{E}$ and $\mathcal{E}^c$, we define the relations $\rightarrow_{\mathcal{E}}$ and $\mapsto$, subsets of $\rightarrow$ as,

Definition of $\rightarrow_{\mathcal{E}}$:

$e \rightarrow_{\mathcal{E}} f$ iff

$$(e, f \in \mathcal{E}) \quad \wedge$$
$$((e \rightsquigarrow f) \vee (e \prec f) \vee (\exists s_1 \in \mathcal{E} \ : \ e \prec s_1 \wedge r_1 \rightarrow_{\mathcal{E}} f)).$$

In other words two events are related by $\rightarrow_{\mathcal{E}}$ iff the causality chain is formed by the events in $\mathcal{E}$.

Definition of $\mapsto$:

$e \mapsto f$ iff,

$$(e, f \in \mathcal{E}) \quad \wedge$$
$$((e \rightsquigarrow f) \vee (e \prec f) \vee (\exists s_1 \in \mathcal{E} \ : \ e \prec s_1 \wedge r_1 \rightarrow f)).$$

The relations is motivated from the fact that a computation restricted to the event set $\mathcal{E}$ is not synchronous if there exists a crown, such that causality chains are formed by the relation $\rightarrow_{\mathcal{E}}$. That is, a computation is sychronous if and only if there is no crown such that,

$$s_0 \rightarrow_{\mathcal{E}} r_1, \cdots, s_{k-2} \rightarrow_{\mathcal{E}} r_{k-1}, s_{k-1} \rightarrow_{\mathcal{E}} r_0.$$

It is easy to see that if there exist a crown (of size k) restricted to the event set $\mathcal{E}$ then there exists a crown,

$$s_0 \mapsto r_1, s_1 \mapsto r_2, \cdots, s_{k-2} \mapsto r_{k-1}, s_{k-1} \mapsto r_0.$$

4.2.1 Priority Rule (PR)

We define a total order among the processes of a distributed system as $P_i < P_j$ iff $i < j$. We also define the function $\mathrm{P} : \mathcal{E} \rightarrow \mathbb{N}$, such that $\mathrm{P}(e) = i$ iff e is an event in process P_i. Based on the function P, any message $(s \rightsquigarrow r)$ can be classified into two types (assuming that a process cannot send a message to itself) :

Type 1 A message to a smaller process, i.e., $\mathrm{P}(s) > \mathrm{P}(r)$, and

Type 2 A message to a bigger process, i.e., $\mathrm{P}(s) < \mathrm{P}(r)$.

The messages of type 1 are committed at the send of the message by the process that is executing the send. The receiver commits the message as soon as it receives the message. In case of the message of type 2, the smaller process sends a request message to the bigger process. The bigger process, when in a position to commit the message, executes the message and sends the message (with the same content) to the smaller process. The smaller process commits the message on receiving. For example,

Type 1 $s \rightsquigarrow r$: As shown in figure 6, the process P_2 commits on the message at event s and process P_1 commits at event r.

Type 2 $\underline{s} \rightsquigarrow \underline{r}$: As shown in figure 6, the process P_2 commits the message at the event r', and the process P_3 commits the message at event s'. Therefore, the process P_2 orders its event such that event e causally precedes the message (or the send of the message $\underline{s} \rightsquigarrow \underline{r}$).

Therefore, every message is committed by the participating process at the send and receive of a message from the bigger process to the smaller process. Keeping this in mind we can assume the system has messages from bigger to smaller processes only. Therefore,

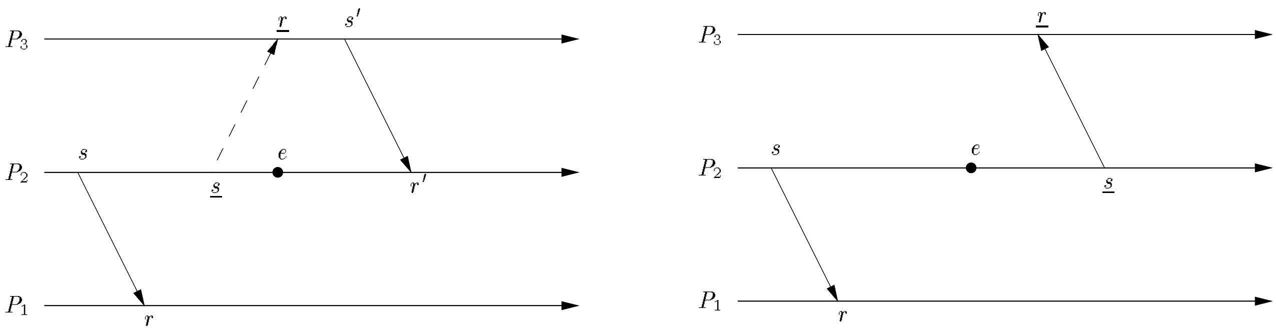
$$\forall (s,r) \in \mathcal{E} \quad : \quad \mathrm{P}(s) > \mathrm{P}(r).$$

4.2.2 Send Condition (SC)

The send condition of a protocol delays the send event. The send condition for message $(s_2, r_2) \in \mathcal{E}$ is formally stated as

$$s_1 \prec s_2 \quad \Longrightarrow r_1 \rightarrow r_2$$

Informally, the condition restricts a process from sending a message until it has the knowledge of receives of all the previous sends. That is, there exists a sequence of messages in $\mathcal{E} \cup \mathcal{E}^c$ such that $r_1 \rightarrow r_2$ holds.



(a) The message along with the underlying message

(b) The resulting message ordering

Figure 6: The messages under the Priority Rule (PR)

4.2.3 Receive Condition (RC)

The receive condition of the delays the receive of a message. It is formally stated for the receive of message $(s_2, r_2) \in \mathcal{E}$ as,

$$s_1 \prec r_2 \quad \Longrightarrow \quad \neg(r_2 \rightarrow r_1).$$

Informally, the condition delays the receive of a message until it is sure that there cannot exist a message in $\mathcal{E} \cup \mathcal{E}^c$ such that $r_2 \rightarrow r_1$.

4.3 Implementation

In this section we describe how acknowledgment messages can be used to satisfy the send and receive conditions.

In the algorithm, every message $e \rightsquigarrow f$ has a underlying acknowledgment message and is represented as $e.ack \rightsquigarrow f.ack$, as shown in figure 7, where $e, f \in \mathcal{E}$ and $e.ack, f.ack \in \mathcal{E}^c$ (the control message). The acknowledgment messages are used to implement the *Send Condition* and the *Receive Condition.*

A process P_i can be in one of the two states: *active* or *passive.* The initial state of every process is *active.* A process changes its state according to the following rules:

- On sending a message, the process changes its state from active to passive,
- On receiving an acknowledgment, the process changes its state from passive to active.

Let us consider a message $e \rightsquigarrow f$, were $e \in P_i$ and $f \in P_j$. On receiving the message at f, process P_j executes a send of an acknowledgment $f.ack$, such that $f \prec f.ack$ and process P_i executes a receive of the acknowledgment $e.ack$, such that $f.ack \rightsquigarrow e.ack$ and $e \prec e.ack$ (see figure 7).

4.3.1 Send Protocol (SP)

The Send Protocol prohibits a process from executing a send event ($s \in \mathcal{E}$) when it is *passive.* For example in figure 7, the process P_i was *active* just before the event e, and *passive* until the event

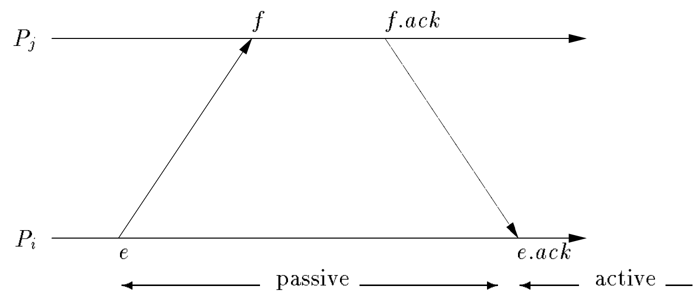


Figure 7: Message with the Acknowledgment

$e.ack$. As the event e can be enabled only if the process is *active*, the last send from the process P_i must have been acknowledged before e.

Formally, we can state SP as,

> To send a message ($s_1 \rightsquigarrow r_1$) from P_i to P_j ($i > j$), wait until it is *active* (i.e., wait for an acknowledgment for the previous send). Therefore,
>
> $$s_1 \prec s_2 \quad \Longrightarrow \quad s_1.ack \prec s_2.$$

Theorem 4 *SP is sufficient to implement SC.*

Proof: Let $s_1 \prec s_2$. We need to show that $r_1 \rightarrow r_2$.
From SP,

$$s_1 \prec s_2 \quad \Longrightarrow \quad s_1.ack \prec s_2. \tag{2}$$

By the conditions of acknowledgment messages, $r_1 \prec r_1.ack$ and $r_1.ack \rightsquigarrow s_1.ack$. Therefore, $r_1 \rightarrow s_1.ack$. From (2), we get that $r_1 \rightarrow s_2$. This in turn implies $r_1 \rightarrow r_2$ since $s_2 \rightsquigarrow r_2$. ||

4.3.2 Receive Protocol (RP)

The Receive Protocol requires that a process send an acknowledgment for the receive of a message only if it is *active* (i.e., it has received an acknowledgment for the previous send). For example, in figure 7 the process P_j can execute the event $f.ack$ when the process is *active*, i.e., the last send from the process P_j has been acknowledged.

Formally, we can state RP as,

> On receiving a message ($s_1 \rightsquigarrow r_1$) from P_i, P_j ($i > j$), commits on the receive of the message and sends an acknowledgment ($r_1.ack \rightsquigarrow s_1.ack$) back when it is *active* (i.e., the acknowledgment of the last send of the message from P_j has been received). Therefore,
>
> $$s_1 \prec r_2 \quad \Longrightarrow \quad s_1.ack \prec r_2.ack.$$

Theorem 5 *If a system satisfies SP, then RP is sufficient to implement RC.*

P_i ::

```
var
    messageQueue    :: Queue of messages
    ackQueue        :: Queue of messages
    state           :: Boolean initially active

□ SendIntent (⟨m, P_i, P_j⟩)
    if (i < j)                              /* message to bigger process PR */
        send (⟨m, P_i, P_j⟩)
    else
        if (state = active)                 /* commit and send the message */
            state = passive
            send (⟨m, P_i, P_j⟩)
            commit (⟨m, P_i, P_j⟩)
        else                                /* wait for the ack message SP */
            Enqueue (messageQueue, ⟨m, P_i, P_j⟩)

□ Receive (⟨m, P_j, P_i⟩)
    if (i > j)                              /* message from smaller process PR */
        SendIntent (⟨m, P_i, P_j⟩)
    else
        Commit (⟨m, P_j, P_i⟩)
        if (state = active)
            send (⟨ack, P_i, P_j⟩)
        else                                /* wait for the previous ack RP */
            Enqueue (ackQueue, ⟨ack, P_i, P_j⟩)

□ Receive (⟨ack, P_i, P_j⟩)
    state = active
    while (notEmpty (ackQueue))             /* send acks if active RP */
        ⟨ack, P_i, P_j⟩ = Dequeue (ackQueue)
        send (⟨ack, P_i, P_j⟩)
    if (notEmpty (messageQueue))
        state = passive
        ⟨m, P_i, P_j⟩ = Dequeue (messageQueue)
        send (⟨m, P_i, P_j⟩)
        commit (⟨m, P_i, P_j⟩)
```

Figure 8: Algorithm to Implement Synchronous Ordering of Messages

Proof: Let $s_1 \prec r_2$. We need to show that $\neg(r_2 \rightarrow r_1)$.
From $s_1 \prec r_2$, we get that $s_1.ack \prec r_2.ack$ using RP. Now consider $s_1.ack$ and r_2. There are two cases possible:

Case 1: $s_1.ack \prec r_2$:

Since $r_1 \rightarrow s_1.ack$ and $s_1.ack \prec r_2$, we get that $r_1 \rightarrow r_2$.

Therefore, $\neg(r_2 \rightarrow r_1)$ holds.

Case 2: $r_2 \prec s_1.ack$:

Since $r_1 \rightarrow s_1.ack$,

$$r_2 \rightarrow r_1 \implies \exists s \in \mathcal{E} \cup \mathcal{E}^c \, : \, r_2 \prec s \prec s_1.ack \, \wedge r \rightarrow r_1.$$

The send $s \notin \mathcal{E}$ otherwise SP is violated. The send $s' \notin \mathcal{E}^c$ otherwise RP is violated. Therefore there cannot exists a send in the interval starting at r_2 and ending at $s_1.ack$.

Therefore, $\neg(r_2 \rightarrow r_1)$ holds.

||

5 Proof of correctness

The correctness of the above algorithm is proved in the usual two steps: safety and liveness.

5.1 Proof of Safety

Our strategy of the proof is to show that if a distributed computation satisfies SC and AC then it will not have a crown formed by causality of events in the set $\mathcal{E}$. We first show that if there exists a crown (of size k) in a distributed run (or computation) then, there also exists a strong crown of size k' where $k' \leq k$. The second part of the proof shows that a synchronous distributed run cannot have a strong crown. We first prove an elementary lemma.

Lemma 1 *If* $(e \mapsto f) \, \wedge \, \neg(e \rightsquigarrow f)$ *then,*

$$(e \prec f) \quad \vee$$

$$\exists (s_1, r_1) \in \mathcal{E} : \, (e \prec s_1 \wedge r_1 \rightarrow f) \qquad \vee \qquad \exists g : \, (e \rightsquigarrow g \wedge g \rightarrow f).$$

Proof: The lemma states that if two events are causally related, then either they are in the same process, or there exists a send event in the same process (and corresponding receive) in the causality chain. ||

The next lemma is used to reduce the crown to a strong crown. If two events $s_1 \in \mathcal{S}$ and $r_2 \in \mathcal{R}$ are causally related as $s_1 \mapsto r_2$, then either both the events are in the same process or there is a send event $s_3 \in \mathcal{E}$ in the causality chain such that $s_1 \prec s_3$.

Lemma 2 *Let* $(s_1 \rightsquigarrow r_1)$ *and* $(s_2 \rightsquigarrow r_2)$. *If* $s_1 \mapsto r_2$ *and SC, then*

$$r_1 \rightarrow r_2 \quad \vee \quad s_1 \prec r_2.$$

Proof: If $s_1 \mapsto r_2$ then (by lemma 1),

$$\begin{array}{rl} (i) & s_1 \prec r_2 \quad \vee \\ (ii) & s_1 \leadsto r_1 \ \wedge \ r_1 \rightarrow r_2 \quad \vee \\ (iii) & \exists s_3 \in \mathcal{E} \ : s_1 \prec s_3 \ \wedge \ r_3 \rightarrow r_2. \end{array}$$

The first two cases directly satisfy the lemma. In the third case, as $s_1 \prec s_3$, therefore

$$r_1 \rightarrow r_3 \qquad\qquad \text{by SC.}$$

and as $r_3 \rightarrow r_2$, we have $r_1 \rightarrow r_2$. ||

We now show that if a crown exists then there also exists a strong crown in the distributed computation.

Lemma 3 *If a distributed computation satisfies SC and RC, and it does not have a strong crown of size 2 formed by the events in $\mathcal{E}$, then it does not have a crown (of size 2) such that,*

$$s_1 \mapsto r_2, \ \ s_2 \mapsto r_1.$$

Proof: Assume that a distributed computation has a crown of size 2.

$$s_1 \mapsto r_2, \ \ s_2 \mapsto r_1.$$

Since the crown is not strong (without loss of generality) assume that $\neg(s_1 \prec r_2)$. From lemma 2, we get that

$$r_1 \rightarrow r_2. \tag{3}$$

From $(r_1 \rightarrow r_2)$ by using RC we get that $\neg(s_2 \prec r_1)$.

Applying lemma 2 again to $\neg(s_2 \prec r_1)$ and $s_2 \mapsto r_1$, we get that

$$r_2 \rightarrow r_1.$$

which is a contradiction to equation 3. ||

Lemma 4 *If a distributed computation satisfying AC and SC has a crown (of size k),*

$$s_0 \mapsto r_1, s_1 \mapsto r_2, \cdots, s_{k-2} \mapsto r_{k-1}, s_{k-1} \mapsto r_0.$$

then it also contains a strong crown of size ($k' \leq k$) such that,

$$s'_0 \prec r'_1, s'_1 \prec r'_2, \cdots, s'_{k-2} \prec r'_{k-1}, s'_{k-1} \prec r'_0.$$

Proof: If $k \leq 2$, then the theorem follows directly from lemma 3. Assume $k > 2$. Pick any part of the crown starting from any index $i \mod k$,

$$s_{i-1} \mapsto r_i, s_i \mapsto r_{i+1} \qquad (*)$$

such that, $\neg(s_i \prec r_{i+1})$, such an i exists otherwise the crown is already strong. Since $s_i \mapsto r_{i+1}$ (by lemma 2),

$$r_i \rightarrow r_{i+1}.$$

Since $s_{i-1} \mapsto r_i$ and $r_i \rightarrow r_{i+1}$, equation (*) can be reduced to

$$s_{i-1} \mapsto r_{i+1}.$$

Therefore, in the sequence ($k > 2$),

$$s_0 \mapsto r_1, s_1 \mapsto r_2, \cdots, s_{k-2} \mapsto r_{k-1}, s_{k-1} \mapsto r_0.$$

If $\neg(s_i \prec r_{i+1})$ then the sequence can be reduced by removing the (s_i, r_i) message. On repeating the process the resulting sequence can be:

- A strong crown of size $k' \leq k$:

$$s'_0 \prec r'_1, s'_1 \prec r'_2, \cdots, s'_{k'-2} \prec r'_{k'-1}, s'_{k'-1} \prec r'_0.$$

- Or the resulting crown is of size 2, such that

$$s_1 \mapsto r_2, \;\; s_2 \mapsto r_1.$$

  By lemma 3, the two crown is strong.

||

Theorem 6 (Safety) *Any distributed run that satisfies SC, AC and PR is synchronous.*

Proof: The proof is by contradiction. Let a distributed run be not synchronous. Then by Theorem 2 there exists a crown. The conditions of lemma 4 are true, i.e., SC and AC. Therefore by lemma 4, there will exist a strong crown,

$$s_0 \prec r_1, s_1 \prec r_2, \cdots, s_{k-2} \prec r_{k-1}, s_{k-1} \prec r_0.$$

From the crown we get

$$\forall i \in \{0, 2, 3, \ldots, k-1)\} \;\; : \;\; \mathrm{P}(s_i) = \mathrm{P}(r_{i+1 \bmod k}), \qquad (*)$$

and (by PR)

$$\forall i \in \{1, 2, 3, \ldots, k-1\} \;\; : \;\; \mathrm{P}(s_i) > \mathrm{P}(r_i). \qquad (**)$$

Combining (*) and (**) we get,

$$\mathrm{P}(r_0) > \mathrm{P}(s_0),$$

which is a contradiction to PR. ||

Theorem 7 (Safety) *Any distributed computation that satisfies SP and RP is synchronous.*

Proof: The proof is by contradiction. Let a distributed run be not synchronous. Then by Theorem 2 there exists a crown. By lemma 4, there will exist a strong crown,

$$s_0 \prec r_1, s_1 \prec r_2, \cdots, s_{k-2} \prec r_{k-1}, s_{k-1} \prec r_0.$$

If $s_i \prec r_{i+1}$ then by RP, $s_i.ack \rightarrow r_{i+1}.ack$. Therefore, from the crown we get

$$\forall i \in \{1, 2, 3, \ldots, (k-1)\} \quad s_i.ack \rightarrow r_{i+1}.ack, \qquad (*)$$

and (by definition)

$$\forall i \in \{1, 2, 3, \ldots, k\} \quad r_i.ack \rightarrow s_i.ack. \qquad (**)$$

Combining (*) and (**) we get,

$$r_1.ack \rightarrow s_k.ack,$$

but according to the strong crown, we have

$$s_k \prec r_1 \quad \Longrightarrow \quad s_k.ack \prec r_1.ack,$$

which is a contradiction. ||

5.2 Proof of Liveness

Theorem 8 (Liveness) *In a distributed computation that satisfies SP, RP and PR, every process P_k that wants to send a message will eventually be able to send it.*

Proof: By induction on k,
Case $k = 1$: The smallest process P_1 does not send any message therefore it is always *active*. It sends an acknowledgment as soon as it gets a message. Therefore on receiving a message ($s \leadsto r$),

$$r \prec r.ack.$$

Now on applying induction, given that k smallest processes eventually be in *active* state, then $(k+1)$th process if *passive* will eventually be *active*. The process P_{k+1} is *passive* at time t if

1. there exists a send of message (s_1, r_1) at time $t_0 < t$ and
2. the process is *passive* between the time interval from t_0 to t.

Therefore, there exists an acknowledgment message $(r_1.ack, s_1.ack)$ from a process P(r_1) to P_{k+1} such that,

1. the acknowledgment is in transit, or
2. send of the acknowledgment will eventually be executed when the process P(r_1) is *active*.

If the message is in transit then process P_{k+1} will eventually receive $s_1.ack$ and become *active*. If the second condition is true, then as P$(r_1) < P_{k+1}$ therefore, P(r_1) will eventually turn active and execute send of acknowledgment message. Therefore, process P_{k+1} will eventually be *active*. ||

6 Conclusions

The algorithm consists of three components: the priority rule, send condition, and the receive condition. The priority rule results in one control message and a delay of less than $2t$ time units (assuming that the upper bound in the delay between any two processes is t time units) for every message $(s, r) \in \mathcal{E}$ if $\mathrm{P}(s) < \mathrm{P}(r)$. In the case of any message $(s, r) \in \mathcal{E}$ where $\mathrm{P}(s) > \mathrm{P}(r)$ there are no control message or delay introduced. However, note that during the delay introduced due to PR, the process can continue to execute any other event.

The send condition introduces only one control message of every message and the receive condition introduces a delay of which is upper bounded by nt, were n is the number of processes in the system. Therefore, the message complexity is 1.5 control messages ($\in \mathcal{E}^c$) for every message $(s, r) \in \mathcal{E}$.

On comparing with the informal algorithm for synchronous ordering, where the sender waits until the receive of an acknowledgment, the resulting algorithm has a higher degree of concurrency. This can be easily seen as both the send condition and the receive conditions are satisfied by informal algorithm.

In this paper we studied the characteristics of a synchronous ordering of messages. The necessary characteristics, i.e., asymmetric and both sender and receiver based protocol, for any algorithm to ensure synchronous ordering were presented. The conditions sufficient (PR, SP, and RP) to implement synchronous ordering were presented based on the necessary characteristics to ensure safety properties. Further, an algorithm was presented based on acknowledgment messages to satisfying SP and RP conditions.

7 Acknowledgments

We would like to thank Alex Tomlinson for his comments and suggestions.

References

[1] Charron-Bost, B., Mattern, F. and Tel, G., "Synchronous and Asynchronous Communication in Distributed Computations", Tech Report TR91.55, LITP, University Paris 7, France, Sept. 1991.

[2] Cormen, T.H., Leiserson, C.E., and Rivest, R.L., *Introduction to Algorithms*, The MIT Press, 1990.

[3] Chandy, K.M. and Lamport, L. "Distributed Snapshots: Determining Global States of Distributed Systems", *ACM Trans., on Computer Systems,* 3(1): 63–75, Feb., 1985.

[4] Birman, K.P., and Joseph, T.A., "Reliable Communication in the Presence of Failures", *ACM Trans. on Computer Systems*, 5(1): 47–76, 1987.

[5] Dijkstra, E.W., Feijen, W.H.J., and van Gasteren, A.J.M., "Derivation of a Termination Detection Algorithm for Distributed Computations", *Inf. Proc. Letters,* 16:217-219, 1983.

[6] Goldman, K. J., "Highly Concurrent Logically Synchronous Multicast", *Proc. of the 3rd International Workshop on Distributed Algorithm*, J-C Bermond, M. Raynal (eds), pp 94–110, Springer-Verlag, 1989.

[7] Joseph, T.A., and Birman, K.P., "Reliable Broadcast Protocol", *Distributed Systems,* Edited by Sape Mullender, pp 294–317, Addison-Wesley, 1989.

[8] Lamport, L., "Time, Clocks and the Ordering of Events in a Distributed System", *Communications of the ACM* 21(7): 95–114, July 1978.

[9] Mattern, F., "Virtual Time and Global States of Distributed Systems", *Proc. Workshop on Parallel and Distributed Algorithms*, M. Cosnard et al. (eds), pp 215–226, Elsevier/North Holland, 1989.

[10] Mostefaoui, A., and Raynal, M., "Causal Multicast in Overlapping Groups: Towards a Low Cost Approach", *Proc. IEEE Int. Conf. on Future Trends of Dist. Comp. Systems.* Lisboa, Sept 1993.

[11] Raynal, M. and Helary, J-M., *Synchronization and Control of Distributed Systems and Programs*, Wiley & Sons, 1990.

[12] Raynal, M., Schiper, A. and Toueg, S., "The Causal Ordering Abstraction and a Simple way to Implement It", *Information Processing Letters*, 39(6): 343–350, 1991.

[13] Schiper, A., Eggli, J., and Sandoz, A., "A New Algorithm to Implement Causal Ordering", *Proc. of the 3rd Int. Workshop on Distributed Algorithms,* pp. 219-232, Springer Verlag, 1989.