# Logical Instantaneity and Causal Order: Two "First Class" Communication Modes for Parallel Computing

Michel Raynal

IRISA
Campus de Beaulieu
35042 Rennes Cedex, France
raynal@irisa.fr

**Abstract.** This paper focuses on two communication modes, namely *Logically Instantaneity* (LI) and *Causal Order* (CO). These communication modes address two different levels of quality of service in message delivery. LI means that it is possible to timestamp communication events with integers in such a way that (1) timestamps increase within each process and (2) the sending and the delivery events associated with each message have the same timestamp. So, there is a logical time frame in which for each message, the send event and the corresponding delivery events occur simultaneously. CO means that when a process delivers a message $m$, its delivery occurs in a context where the receiving process knows all the causal past of $m$. Actually, LI is a property strictly stronger than CO.
The paper explores these noteworthy communication modes. Their main interest lies in the fact that they deeply simplify the design of message-passing programs that are intended to run on distributed memory parallel machines or cluster of workstations.

**Keywords**: Causal Order, Cluster of Workstations, Communication Protocol, Distributed Memory, Distributed Systems, Logical Time, Logical Instantaneity, Rendezvous.

## 1 Introduction

Designing message-passing parallel programs for distributed memory parallel machines or clusters of workstations is not always a trivial task. In a lot of cases, it reveals to be a very challenging and error-prone task. That is why any system designed for such a context has to offer the user a set *Services* that simplify his programming task. The ultimate goal is to allow him to concentrate only on the problem he has to solve and not the technical details of the machine on which the program will run.

Among the services offered by such a system to upper layer application processes, *Communication Services* are of crucial importance. A communication service is defined by a pair of matching primitives, namely a primitive that allows to send a message to one or several destination processes and a primitive

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 35–42, 2000.

that allows a destination process to receive a message sent to it. Several communication services can coexist within a system. A communication service is defined by a set of properties. From a user point of view, those properties actually define the *quality of service* (*QoS*) offered by the communication service to its users. These properties usually concern reliability and message ordering.

A *reliability* property states the conditions under which a message has to be delivered to its destination processes despite possible failures. An *ordering* property states the order in which messages have to be delivered; usually this order depends on the message sending order. FIFO, causal order (CO) [4,14] and *total order* (TO) [4] are the most encountered ordering properties [7]. Reliability and ordering properties can be combined to give rise to powerful communication primitives such as Atomic Broadcast [4] or Atomic Multicast to asynchronous groups

Another type of communication service is offered by CSP-like languages. This communication type assumes reliable processes and provides the so-called *rendezvous* (RDV) communication paradigm [2,8] (also called *synchronous* communication.) "A system has synchronous communications if no message can be sent along a channel before the receiver is ready to receive it. For an external observer, the transmission then looks like instantaneous and atomic. Sending and receiving a message correspond in fact to the same event" [5]. Basically, RDV combines synchronization and communication. From an operational point of view, this type of communication is called *blocking* because the sender process is blocked until the receiver process accepts and delivers the message. "While asynchronous communication is less prone to deadlocks and often allows a higher degree of parallelism (...) its implementation requires complex buffer management and control flow mechanisms. Furthermore, algorithms making use of asynchronous communication are often more difficult to develop and verify than algorithms working in a synchronous environment" [6]. This quotation expresses the relative advantages of synchronous communication with respect to asynchronous communication.

This paper focuses on two particular message ordering properties, namely, *Logical Instantaneity* (LI), and *Causal Order* (CO). The LI communication mode is weaker than RDV in the sense that it does not provide synchronization; more precisely, the sender of a message is not blocked until the destination processes are ready to deliver the message. But LI is stronger than CO (*Causally Ordered* communication). CO means that, if two sends are causally related [10] and concern the same destination process, then the corresponding messages are delivered in their sending order [4]. Basically, CO states that when a process delivers a message $m$, its delivery occurs in a context where the receiving process already knows the causal past of $m$. CO has received a great attention in the field of distributed systems, this is because it greatly simplifies the design of protocols solving consistency-related problems [14].

It has been shown that these communication modes form a strict hierarchy [6,15]. More precisely, RDV $\Rightarrow$ LI $\Rightarrow$ CO $\Rightarrow$ FIFO, where X $\Rightarrow$ Y means that if the communications satisfy the X property, they also satisfy the Y property. (More sophisticated communication modes can found in [1].) Of course, the less

constrained the communications are, the more efficient the corresponding executions can be. But, as indicated previously, a price has to be paid when using less constrained communications: application programs can be more difficult to design and prove, they can also require sophisticated buffer management protocols. Informally, LI provides the illusion that communications are done according to RDV, while actually they are done asynchronously. More precisely, LI ensures that there is a *logical* time frame with respect to which communications are synchronous.

This paper is mainly centered on the definition of the LI and CO communication modes. It is composed of four sections. Section 2 introduces the underlying system model. Then, Section 3 and Section 4 glance through the LI and CO communication modes, respectively. As a lot of literature has been devoted to CO, the paper content is essentially focused on LI.

## 2 Underlying System Model

### 2.1 Underlying Asynchronous Distributed System

The underlying asynchronous distributed system consists of a finite set $P$ of $n$ processes $\{P_1, \ldots, P_n\}$ that communicate and synchronize only by exchanging messages. We assume that each ordered pair of processes is connected by an asynchronous, reliable, directed logical channel whose transmission delays are unpredictable but finite[1]. The capacity of a channel is supposed to be infinite. Each process runs on a different processor, processors do not share a common memory, and there is no bound on their relative speeds.

A process can execute internal, send and receive operations. An internal operation does not involve communication. When $P_i$ executes the operation $send(m, P_j)$ it puts the message $m$ into the channel connecting $P_i$ to $P_j$ and continues its execution. When $P_i$ executes the operation $receive(m)$, it remains blocked until at least one message directed to $P_i$ has arrived, then a message is withdrawn from one of its input channels and delivered to $P_i$. Executions of internal, send and receive operations are modeled by internal, sending and receive events. Processes of a distributed computation are *sequential*; in other words, each process $P_i$ produces a *sequence* of events $e_{i,1} \ldots e_{i,s} \ldots$ This sequence can be finite or infinite. Moreover, processes are assumed to be reliable.

Let $H$ be the set of all the events produced by a distributed computation. This computation is modeled by the partially ordered set $\widehat{H} = (H, \stackrel{hb}{\rightarrow})$, where $\stackrel{hb}{\rightarrow}$ denotes the well-known Lamport's *happened-before* relation [10]. Let $e_{i,x}$ and $e_{j,y}$ be two different events:

$$e_{i,x} \stackrel{hb}{\rightarrow} e_{j,y} \Leftrightarrow \begin{cases} i = j \wedge x < y \\ \vee\ \exists m : e_{i,x} = send(m, P_j) \wedge e_{j,y} = receive(m) \\ \vee\ \exists e : e_{i,x} \stackrel{hb}{\rightarrow} e \wedge e \stackrel{hb}{\rightarrow} e_{j,y} \end{cases}$$

[1] Note that channels are not required to be FIFO.

So, the underlying system model is the well known reliable asynchronous distributed system model.

### 2.2 Communication Primitives at the Application Level

The communication interface offered to application processes is composed of two primitives denoted SEND and DELIVER.

- The SEND$(m, dest_m)$ primitive allows a process to send a message $m$ to a set of processes, namely $dest_m$. This set is defined by the sender process $P_i$ (without loss of generality, we assume $P_i \notin dest_m$). Moreover, every message $m$ carries the identity of its sender: $m.sender = i$. The corresponding application level event is denoted SEND$_{m.sender}(m)$.
- The DELIVER$(m)$ primitive allows a process (say $P_j$) to receive a message that has been sent to it by an other process (so, $P_j \in dest_m$). The corresponding application level event is denoted DELIVER$_j(m)$.

It is important to notice that the SEND primitive allows to multicast a message to a set of destination processes which is dynamically defined by the sending process.

## 3 Logically Instantaneous Communication

### 3.1 Definition

In the context of LI communication, when a process executes SEND$(m, dest_m)$ we say that it "LI-sends" $m$. When a process executes DELIVER$(m)$ we say that it "LI-delivers" $m$. Communications of a computation satisfy the LI property if the four following properties are satisfied.

- *Termination.* If a process LI-sends $m$, then $m$ is made available for LI-delivery at each process $P_j \in dest_m$. $P_j$ effectively LI-delivers $m$ when it executes the corresponding DELIVER primitive[2].
- *Integrity.* A process LI-delivers a message $m$ at most once. Moreover, if $P_j$ LI-delivers $m$, then $P_j \in dest_m$.
- *Validity.* If a process LI-delivers a message $m$, then $m$ has been LI-sent by $m.sender$.
- *Logical Instantaneity.* Let $\mathbb{N}$ be the set of natural integers. This set constitutes the (logical) time domain. Let $H_a$ be the set of all application level communication events of the computation. There exists a timestamping function $\mathcal{T}$ from $H_a$ into $\mathbb{N}$ such that $\forall (e, f) \in H_a \times H_a$ [11]:

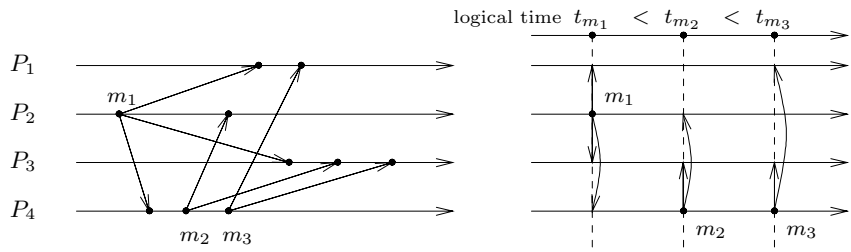
[2] Of course, for a message (that has been LI-sent) to be LI-delivered by a process $P_j \in dest_m$, it is necessary that $P_j$ issues "enough" invocations of the DELIVER primitive. If $m$ is the $(x+1)$th message that has to be LI-delivered to $P_j$, its LI-delivery at $P_j$ can only occur if $P_j$ has first LI-delivered the $x$ previous messages and then invokes the DELIVER primitive.

**(LI$_1$)** $e$ and $f$ have been produced by the same process with $e$ first
$\Rightarrow \mathcal{T}(e) < \mathcal{T}(f)$

**(LI$_2$)** $\forall m : \forall j \in dest_m : e = \text{SEND}_{m.sender}(m) \wedge f = \text{DELIVER}_j(m)$
$\Rightarrow \mathcal{T}(e) = \mathcal{T}(f)$

From the point of view of the communication of a message $m$, the event $\text{SEND}_{m.sender}(m)$ is the cause and the events $\text{DELIVER}_j(m)$ ($j \in dest_m$) are the effects. The termination property associates effects with a cause. The validity property associates a cause with each effect (in other words, there are no spurious messages). Given a cause, the integrity property specifies how many effects it can have and where they are produced (there are no duplicates and only destination processes may deliver a message). Finally, the logical instantaneity property specifies that there is a logical time domain in which the send and the deliveries events of every message occur at the same instant.

Figure 1.a describes communications of a computation in the usual space-time diagram. We have: $m_1.sender = 2$ and $dest_{m_1} = \{1, 3, 4\}$; $m_2.sender = m_3.sender = 4$, $dest_{m_2} = \{2, 3\}$ and $dest_{m_3} = \{1, 3\}$. These communications satisfy the LI property as shown by Figure 1.b. While RDV allows only the execution of Figure 1.b, LI allows more concurrent executions such as the one described by Figure 1.a.




a. A "Real" Computation b. Its "LI" Counterpart

**Fig. 1.** Logical Instantaneity

### 3.2 Communication Statements

In the context of LI communication, two types of statements in which communication primitives can be used by application processes are usually considered.

– *Deterministic Statement.* An application process may invoke the DELIVER primitive and wait until a message is delivered. In that case the invocation appears in a deterministic context (no alternative is offered to the process in case the corresponding SEND is not executed). In the same way, an application process may invoke the SEND primitive in a deterministic context.

– *Non-Deterministic Statement.* The invocation of a communication primitive in a deterministic context can favor deadlock occurrences (as it is the case, for example, when each process starts by invoking DELIVER.) In order to help applications prevent such deadlocks, we allow processes to invoke communication primitives in a non-deterministic statement (ADA and similar languages provide such non-deterministic statements). This statement has the following syntactical form:

**select_com** SEND$(m, dest_m)$ **or** DELIVER$(m')$ **end_select_com**

This statement defines a non-deterministic context. The process waits until one of the primitives is executed. The statement is terminated as soon as a primitive is executed, a flag indicates which primitive has been executed. Actually, the choice is determined at runtime, according to the current state of communications.

### 3.3 Implementing LI Communication

Due to space limitation, there is not enough room to describe a protocol implementing the LI communication mode. The interested reader is referred to [12] where a very general and efficient protocol is presented. This protocol is based on a three-way handshake.

## 4 Causally Ordered Communication

### 4.1 Definition

In some sense Causal Order generalizes FIFO communication. More precisely, a computation satisfies the CO property if the following properties are satisfied:

– *Termination.* If a process CO-sends $m$, then $m$ is made available for CO-delivery at each process $P_j \in dest_m$. $P_j$ effectively CO-delivers $m$ when it executes the corresponding DELIVER primitive.
– *Integrity.* A process CO-delivers a message $m$ at most once. Moreover, if $P_j$ CO-delivers $m$, then $P_j \in dest_m$.
– *Validity.* If a process CO-delivers a message $m$, then $m$ has been CO-sent by $m.sender$.
– *Causal Order.* For any pair of message $m1$ and $m2$ such that CO-$send(m1) \stackrel{hb}{\rightarrow}$ CO-$send(m2)$ then, $\forall\ p_j \in dest_{m1} \cap dest_{m2}$, $p_j$ CO-delivers $m1$ before $m2$.

Actually, CO constraints the non-determinism generated by the asynchrony of the underlying system. It forces messages deliveries to respect the causality order of their sendings.

Figure 2 depicts two distributed computation where messages are broadcast. Let us first look at the computation on the left side. We have $send(m1) \stackrel{hb}{\rightarrow} send(m2)$; moreover, $m_1$ and $m_2$ are deliverd in this order by each process. The sending of $m_3$ is causally related to neither $m_1$ nor $m_2$, hence no constraint

applies to its delivery. If follows that communication of this computation satisfies CO property. The reader can easily verify that the right computation does not satisfy the CO communication mode (the third process delivers $m_1$ after $m_2$, while their sendings are ordered the other way by $\stackrel{hb}{\rightarrow}$).

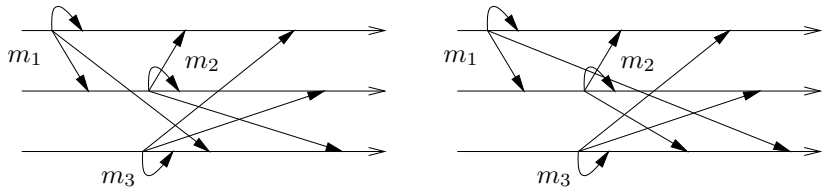



**Fig. 2.** Causal Order

### 4.2 Implementation Protocols

Basically, a protocol implementing causal order associates with each message a delivery condition. This condition depends on the current context of the receiving process (i.e., which messages it has already delivered) an on the context of the message (i.e., which message have been sent in the causal past of its sending).

The interested reader will find a basic protocol implementing causal order in [14]. More efficient protocols can be found in [3] for broadcast communication and in [13] for the general case (multicast to arbitrary subsets of processes). A formal (ad nice) study of protocols implementing the CO communication mode can be found in [9].

## References

1. Ahuja M. and Raynal M., An implementation of Global Flush Primitives Using Counters. *Parallel Processing Letters*, Vol. 5(2):171-178, 1995.
2. Bagrodia R., Synchronization of Asynchronous Processes in CSP. *ACM TOPLAS*, 11(4):585-597, 1989.
3. Baldoni R., Prakash R., Raynal M. and Singhal M., Efficient $\Delta$-Causal Broadcasting. *Journal of Computer Systems Science and Engineering*, 13(5):263-270, 1998.
4. Birman K.P. and Joseph T.A., Reliable Communication in the Presence of Failures. *ACM TOCS*, 5(1):47-76, 1987.
5. Bougé L., Repeated Snapshots in Distributed Systems with Synchronous Communications and their Implementation in CSP. *TCS*, 49:145-169, 1987.
6. Charron-Bost B., Mattern F. and Tel G., Synchronous, Asynchronous and Causally Ordered Communications. *Distributed Computing*, 9:173-191, 1996.
7. Hadzilacos V. and Toueg S., Reliable Broadcast and Related Problems. In *Distributed Systems*, ACM Press (S. Mullender Ed.), New-York, pp. 97-145, 1993.

8. Hoare C.A.R., Communicating Sequential Processes. *Communications of the ACM*, 21(8):666-677, 1978.
9. Kshemkalyani A.D. and Singhal M., Necessary and Sufficient Conditions on Information for Causal Message Ordering and their Optimal Implementation. *Distributed Computing*, 11:91-111, 1998.
10. Lamport, L., Time, Clocks and the Ordering of Events in a Distributed System, *Communications of the ACM*, 21(7):558-565, 1978.
11. Murty V.V. and Garg V.K., Synchronous Message Passing. *Tech. Report* TR ECE-PDS-93-01, University of Texas at Austin, 1993.
12. Mostefaoui A., Raynal M. and Verissimo P., Logically Instantaneous Communications in Asynchronous Distributed Systems. *5th Int. Conference on Parallel Computing Technologies (PACT'99)*, St-Petersburg, Springer Verlag LNCS 1662, pp. 258-270, 1999.
13. Prakash R., Raynal M. and Singhal M., An adaptive Causal Ordering Algorithm Suited to Mobile Computing Environments. *Journal of Parallel and Distributed Computing*, 41:190-204, 1997.
14. Raynal M., Schiper A. and Toueg S., The Causal ordering Abstraction and a Simple Way to Implement it. *Information Processing Letters*, 39:343-351, 1991.
15. Soneoka T. and Ibaraki T., Logically Instantaneous Message Passing in Asynchronous Distributed Systems. *IEEE TC*, 43(5):513-527, 1994.