The Logically Instantaneous Communication Mode: a Communication Abstraction *

Achour Mostéfaoui[a] and Michel Raynal[a] and Paulo Veríssimo[b]

[a]IRISA - Campus de Beaulieu, 35042 Rennes Cedex, France
{mostefaoui,raynal}@irisa.fr

[b]University of Lisboa, C5 Campo Grande, 1700 Lisboa, Portugal
pjv@di.fc.ul.pt

Communication is *Logically Instantaneous* (LI) when there is a logical time frame in which for each message, the send event and the corresponding delivery events occur simultaneously. LI is stronger than Causally Ordered (CO) communication, but weaker than Rendezvous (RDV) communication. This paper explores *Logically Instantaneous* communication and provides a simple and efficient protocol that implements LI on top of asynchronous distributed systems.

Keywords: Asynchronous Distributed System, Communication Protocol, Logical Time, Logically Instantaneous Communication, Rendezvous Communication.

1. Introduction

Applications supported by distributed systems are usually difficult to design and implement. This is mainly due to the fact that those systems are asynchronous and prone to failures. So, one of the main issues distributed designers have to solve consists in providing applications with high level communication primitives that offer a "good" quality of service. "Good" means here that these primitives must offer *synchronization/ordering* guarantees (to master asynchronism) and *reliability* guarantees (to face failures). It is important to note that these issues make distributed computing very different from parallelism or realtime. Actually, *Parallelism* aims to produce efficient computations. Satisfaction of *Realtime* constraints allows computations to produce their results on time. This paper presents a communication service addressing synchronization guarantees. The reader interested by reliability guarantees is referred to [1,2].

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 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 an *ordering synchronization* property that states the order (if any) in which messages have to be delivered. Usually the ordering property depends on the message sending order. FIFO, causal

*A previous version of this paper has appeared in [1] order (CO) [3,4] and *total order* (TO) [3,5] are the most encountered ordering properties [6].

Another type of communication service is offered by CSP-like languages (*e.g.*, the *Occam* distributed programming language). This communication type assumes reliable processes and provides the so-called *rendezvous* (RDV) communication paradigm [7–9] (also called *synchronous* communication.) This communication property is a synchronization property. "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" [10]. 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" [11]. This quotation expresses the relative advantages of synchronous communication with respect to asynchronous communication.

This paper focuses on a particular message ordering property, namely, *Logical Instantaneity* (LI [12]). This property 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 [13] and concern the same destination process, then the corresponding messages are delivered in their sending order [3]. CO has received a great attention as it simplifies the design of protocols solving consistency-related problems [4]. It has been shown that these communication modes form a strict hierarchy [11,12]. 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. 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. As a very simple example let us consider a system made of only two processes p and q plus a single directed channel from p to q. If the channel is FIFO, then it ensures the LI property: it is possible to build a logical time such that matching send and receive events occur at the same date, *i.e.*, at the same logical instant. When considering (non realtime) distributed programs, LI can not be distinguished from RDV. So, any program designed with the assumption that communications satisfy RDV can actually be run with LI communications. This is particularly interesting and suggests the following approach: first design (and prove) programs assuming RDV communications; and then run the obtained programs on top of a system offering LI communications (we only consider here non realtime programs). So, the LI communication paradigm can benefit to high performance computing when applications are run top on distributed memory parallel machines. In that sense this paper contributes to the definition of a communication service that can be used to provide efficient executions of Rendezvous-based parallel algorithms.

This paper explores the LI communication paradigm. It is composed of five sections. Section 2 introduces the underlying asynchronous distributed system model and defines logically instantaneous communication. Then, Section 3 presents a protocol implementing LI communication on top of an asynchronous distributed system. Section 4 proves the protocol correction. Finally, Section 5 provides a few concluding remarks.

2. Asynchronous Distributed Systems and LI Communication

2.1. Underlying Asynchronous Distributed System

The underlying asynchronous distributed system consists of a finite set P of n processes $\{P_1, \dots, P_n\}$ 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 (channels are not required to be FIFO). 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 $e_{i,1} \dots e_{i,s} \dots$ This sequence can be finite or infinite. Moreover, processes are assumed to be reliable, *i.e.*, they do not suffer failures (Section 5 will consider process crash failures).

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 [13]. Let $e_{i,x}$ and $e_{j,y}$ be two different events:

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

So, the underlying system model is the well known reliable asynchronous distributed system model. Our aim is to build on top of it a distributed system whose communications satisfy the LI property.

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

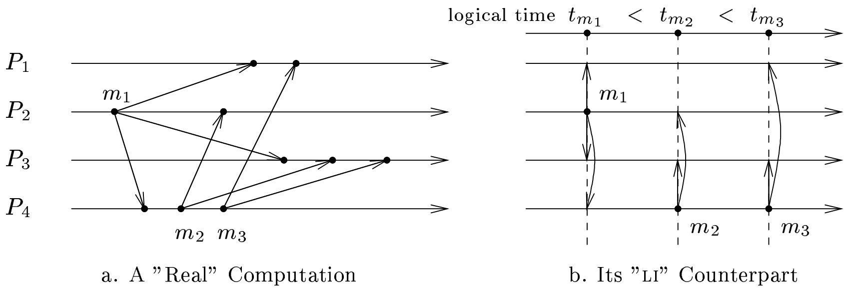

Figure 1. Logical Instantaneity

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 $\text{SEND}_{m.sender}(m)$.

- 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 $\text{DELIVER}_j(m)$.

Previous proposals [14,12] consider a message is sent to a single destination process. Here, the SEND primitive allows to multicast a message to an arbitrary set of destination processes (this set is defined online by the sending process).

2.3. 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 $I\!N$ 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 TF from H_a into $I\!N$ such that $\forall (e, f) \in H_a \times H_a$ [14]:

[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.

(LI$_1$) e and f have been produced by the same process with e first
$\Rightarrow TF(e) < TF(f)$

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

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.

2.4. Communication Statements

We consider two types of statements in which communication primitives can be used by application processes.

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 [15] (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. A Protocol Implementing LI Communication

3.1. Underlying Principle

The protocol is based on a few simple ideas. First, a message sending is processed as a message delivery. Second, a Lamport timestamp (local clock value, sender identity) is associated with each message and messages are delivered according to their timestamp order [13]. Finally, after having initiated a message sending, its sender has to commit or to abort it.

The computation of the timestamp associated with a message m requires the cooperation of its sender and of its destination processes. This cooperation is done in a classical way: each process proposes a timestamp for m and the greatest proposal becomes the final timestamp of m. (This cooperation scheme is known as "Skeen's protocol", as described in [3].)

At the protocol level, a message m is made of several fields. Those are:
- $m.id$: the identity of m. All message identities are assumed to be distinct.
- $m.sender$: the identity of the sender (as seen before).
- $m.content$: the content of m (defined by the sender process).
- $m.ts$: the current timestamp associated with m.
- $m.final\_ts$: whose value is $true$ or $false$. $m.final\_ts = true$ means that m has got its final timestamp; $m.final\_ts = false$ means that the current timestamp of m is not necessarily its final timestamp, it is only a timestamp proposal.

Implementing a LI_communication mode means implementing a timestamping function TF respecting the two ordering properties given in section 2.3. Our protocol implements such a function by associating with each message m a pair $(m.ts, m.sender)$ according to which the queues are sorted. Let $m1$ and $m2$ be two messages in $queue_i$. The message $m1$ precedes the message $m2$ if $(m1.ts, m1.sender) < (m2.ts, m2.sender)$ (lexicographic order defined in [13]).

Each process P_i manages a queue ($queue_i$) where it stores the messages that are currently candidates to be delivered at P_i. The primitive *append* allows to add a message into $queue_i$. The primitive *remove* allows to suppress a message from $queue_i$. This queue is assumed to be sorted according to the current timestamps of the messages it contains. The primitive *head* returns the message that is currently the first of $queue_i$ (*i.e.*, the one with the smallest timestamp). To simplify the protocol description, we assume that $queue_i$ is never empty: it always contains a "sentinel" message m characterized by: $m.ts = +\infty$ and $m.final\_ts = false$. (As we will see in the protocol description, this message will never obtains a final timestamp, and consequently will never be delivered.)

Three procedures (Section 3.2) and a task T_i (Section 3.3) implements the protocol at each process P_i. A procedure implements a communication primitive in a given context: SEND called in a deterministic context, DELIVER called in a deterministic context, and the SEND/DELIVER non-deterministic statement. The task T_i manages the local data structure $queue_i$. It constitutes the core of the protocol by executing operations required by the three procedures and handling messages received from other processes.

3.2. Implementing the Primitives
SEND in a deterministic context

The primitive SEND is implemented by the procedure given below, where $sent\_msg$ and msg_dest denote the data value of the message and the set of its destination processes, respectively.

Procedure SEND($sent\_msg, msg\_dest$)

create a new message identity msg_id;
add_to_queue($msg\_id, msg\_content, msg\_dest$);
wait until $head(queue_i).id = msg\_id \wedge head(queue_i).final\_ts$;
commit(msg_id); /* Event $\text{SEND}_i(m)$ */
remove $head(queue_i)$ **from** $queue_i$

After having defined a new message identity, P_i asks the task T_i to add this new message into $queue_i$. Then, it waits until this message becomes the head of the queue and has got its final timestamp. When this occurs, P_i commits the sending of the message: this means the event $\text{SEND}_{m.sender}(m)$ occurs. Then, P_i suppresses m from $queue_i$.

DELIVER in a deterministic context

DELIVER in a deterministic context is implemented by the procedure given below. P_i waits until the head of the queue contains a message that has got its final timestamp. When this occurs, the content of the message is delivered to P_i and the message is removed from the queue.

Procedure DELIVER(del_msg)
wait until $head(queue_i).final\_ts$;
let m **be** $head(queue_i)$;
$del\_msg \leftarrow m.content$; /* Event $\text{DELIVER}_i(m)$ */
remove m **from** $queue_i$

SEND/DELIVER in a non-deterministic context

This non-deterministic context communication statement is implemented by the function given below. First P_i initiates the sending of the message (so, the first two lines are the same as in the primitive SEND. Then, P_i waits until there is a message m with a final timestamp at the head of its queue. If m is the message it has just tried to send, then the SEND is successful: it is committed and "sending" is provided as a result of the call to the non-deterministic statement. If m is another message, then the DELIVER is successful: P_i aborts the sending of the message it has initiated, delivers the content of m and provides "delivery" as a result of the call to the non-deterministic statement.

function SEND($sent\_msg, msg\_dest$)_or_DELIVER(del_msg)
create a message identity msg_id;
add_to_queue($msg\_id, sent\_msg, msg\_dest$);
wait until $head(queue_i).final\_ts$;
let m **be** $head(queue_i)$;
case $m.id = msg\_id$ $\rightarrow$ **commit**(msg_id);
return "sending" /* Event $\text{SEND}_i(m)$ */
$m.id \neq msg\_id$ $\rightarrow$ **abort**(msg_id);
$del\_msg \leftarrow m.content$;
return "delivery" /* Event $\text{DELIVER}_i(m)$ */
endcase;
remove m **from** $queue_i$

3.3. Queue Management

As indicated previously, the management of the local data structure $queue_i$ constitutes the core of the protocol. It is realized by a task T_i that runs in the background of P_i. Each set of statements of T_i is executed atomically. The work of T_i is [16]:

- The implementation of the local calls to **add_to_queue**, **commit** and **abort** issued by P_i. This entails the sending of control messages to peer tasks of other processes.
- The processing of the control messages received from the other tasks. These control messages are TS_REQUEST, TS_PROPOSAL, COMMITTED and ABORTED. The two first are related to the final timestamp determination of messages. The third one is used to convey the final timestamp associated with a message. Finally, the last one is related to the the implementation of the non-deterministic communication statement.

In addition to $queue_i$, the task T_i manages three variables. The first one is h_i, a local Lamport clock handled in the usual way [13]. The other two, namely, $dest_i$ and $wait\_ts\_from_i$, are relevant only when P_i is engaged in a SEND(m) primitive (deterministic or not). They have the following meaning. The variable $dest_i$ is used to memorize the identities of the processes that are the destinations of m (namely, $dest_m$). The variable $wait\_ts\_from_i$ (which is initialized to $dest_i$) is updated ($wait\_ts\_from_i \leftarrow wait\_ts\_from_i - \{j\}$) when a timestamp proposal (TS_PROPOSAL) for m is received from P_j; so $wait\_ts\_from_i = \emptyset$ indicates that T_i has received all the timestamp proposals for the message m it wants to send: this means that m has got its final timestamp.

The behavior of T_i is the following:

When **add_to_queue** is called (deterministic or non deterministic send primitive), T_i builds a message with the appropriate fields, and adds it to $queue_i$. Then, it sends to each destination process a TS_REQUEST control message asking it to send back a timestamp proposal for m. It also appropriately updates the local control variables $dest_i$ and $wait\_ts\_from_i$.

When a task receives a TS_REQUEST control message concerning a message m, it first builds a corresponding entry in its queue, and then answers by proposing a timestamp for m.

When, T_i receives a timestamp proposal for a message m, it updates the timestamp of m (namely, $m.ts$) to the max of its current value and the proposed value. If all the timestamp proposals have been received, then m has its final timestamp (the maximum of all proposed timestamps), consequently, $m.final\_ts$ is set to $true$. From now on, the message is “ready” to be sent (if P_i is its sender) or delivered (if P_i is a destination process). The message is “delayed” until either it is at the head of the queue and is required by P_i, or its sending is aborted by its sender (non-deterministic primitive).

If the sending process commits the sending of m, then a COMMITTED control message is sent to all the processes of $dest_m$. When a destination receives this message, it provides its copy of the message with the final timestamp and makes the message ready to be locally delivered. The message will be locally delivered when it will be at the head of the local queue and will be required by the destination process.

If the sender process aborts the sending of m (non-deterministic primitive), then an

ABORTED control message is sent to all the processes of $dest_m$. When a destination process receives this message it destroys its copy of m.

```
Task T_i
  h_i ← 0;
  cobegin
    || upon a local call to add_to_queue(msg_id, msg_content, msg_dest)
         h_i ← h_i + 1; dest_i ← msg_dest;
         build a message structure m such that
              m.sender ← i, m.ts ← h_i, m.id ← msg_id,
              m.final_ts ← false, m.content ← msg_content;
         append m to queue_i;
         ∀k ∈ dest_i send TS_REQUEST(m.id, m.content) to P_k;
         wait_ts_from_i ← msg_dest

    || upon reception of TS_REQUEST(msg_id, msg_content) from P_j
         /* first time P_i hears about this message */
         h_i ← h_i + 1;
         build a message structure m such that
              m.sender ← j, m.ts ← h_i, m.id ← msg_id,
              m.final_ts ← false, m.content ← msg_content;
         append m to queue_i;
         send TS_PROPOSAL(m.id, h_i) to m.sender

    || upon reception of TS_PROPOSAL(msg_id, msg_ts) from P_j
         if ∃ m ∈ queue_i such that m.id = msg_id then
              m.ts ← max(m.ts, msg_ts);
              wait_ts_from_i ← wait_ts_from_i − {j};
              m.final_ts ← (wait_ts_from_i = ∅);
              h_i ← max(h_i, m.ts)
         endif /* (m ∉ queue_i) ⇒ (the sending of m has been aborted by P_i) */

    || upon a local call to commit(msg_id)
         let m ∈ queue_i such that m.id = msg_id;
         ∀k ∈ dest_i send COMMITTED(m.id, m.ts) to P_k

    || upon reception of COMMITTED(msg_id, msg_ts)
         let m ∈ queue_i such that m.id = msg_id;
         m.ts ← msg_ts; /* the send has been committed */
         m.final_ts ← true;
         h_i ← max(h_i, m.ts)

    || upon a local call to abort(msg_id)
         let m ∈ queue_i such that m.id = msg_id;
         ∀k ∈ dest_i send ABORTED(m.id) to P_k;

    || upon reception of ABORTED(msg_id)
         wait until ∃m ∈ queue_i such that m.id = msg_id;
```

```
(37)        /* The wait is required because channels are not necessarily FIFO: */
(38)        /* ABORTED(msg_id) can arrive before */
(39)        /* the corresponding TS_REQUEST(msg_id, −) */
(40)        remove m from queue_i /* the send has been aborted */
coend
```

3.4. Cost of the Protocol

For each application message, the proposed protocol is actually a three-way handshake protocol (defined by the sequence of control messages: TS_REQUEST, TS_PROPOSAL, COMMITTED/ABORTED). The protocol is *minimal* in the sense that, for each message, it involves only its sender and its destination processes, no other process has to participate in the protocol.

Let d_m be the number of destination processes of m and assume all control messages take one time unit. For each application message, the protocol costs $3d_m$ messages and 3 time units. If the message is aborted, these costs actually are upper bounds. In the broadcast case (all application messages are sent to all processes), the protocol come very close to Lamport's mutual exclusion protocol [13] and has the same message cost, namely $3(n-1)$.

Let us remark that line 11 of the protocol can be removed. In fact, it has been added only to ease the explanation of the correctness proof. The removal of this line permits to have logical clocks that grow much more slowly.

4. Proof

4.1. Integrity

vspace3mm

Proof (Integrity) The integrity property (a message is LI-delivered at most once) follows from the fact that a message that has been LI-delivered is immediately removed from the queue and its sender also removes it from its local queue just after the message got its final timestamp. $\square_{Integrity}$

4.2. Validity

vspace3mm

Lemma 1 *An aborted message will never be delivered to any process and will be eventually removed from the queue of its sender and from the queues of all its destination processes.*

Proof When the function SEND_OR_DELIVER aborts the sending of a message the procedure **abort** is invoked instead of the procedure **commit** and the message is removed from the queue of its sender. Thus, the destination processes of the aborted message will never get the final timestamp for this message (the final timestamp is sent at line 26 when **commit** is invoked). Consequently the aborted message will never be deliverable. At line 34 (procedure **abort**), a message ABORTED is sent to all the destination processes. Due to possibly non-FIFO channels the TS_REQUEST message sent by the sender of m when the non-deterministic send statement executed a call to **add_to_queue** (line 7) may arrive after the ABORTED message. As channels are reliable, both messages will eventually arrive. This is why the destination process first waits for TS_REQUEST (line 35) before removing the aborted messages from the queue (line 40). $\Box_{Lemma\ 1}$

Proof (Validity) The validity property (if a message is LI-delivered, it has been LI-sent) follows from Lemma 1 and the fact that no application message is coined by the protocol. The only message that is coined by the protocol is a control message: the "sentinel" m such that: $m.ts = +\infty$ and $m.ts\_final = false$. This message is used to simplify the protocol description by ensuring the local queues are never empty. As its ts_final field is never set to $false$ this message cannot be delivered. Moreover, by Lemma 1, if the sending of a message is aborted it will never be delivered to any process. $\Box_{Validity}$

4.3. Termination

The termination property states that if the event SEND$_{m.sender}(m)$ is produced, then m is made available for LI-delivery at each destination process.

Lemma 2 *Each sent and non-aborted message will eventually get its final timestamp.*

Proof When a message is proposed for sending either in a deterministic or in a non-deterministic context, **add_to_queue** is invoked and then a wait statement is executed. **add_to_queue** is non-blocking and during its execution (line 7) a TS_REQUEST message is sent to all the destination processes of the message and initializes the variable $wait\_ts\_from$ to the set of all these destination processes identities. Upon the reception of a TS_REQUEST message, a process replies by sending a TS_PROPOSAL. The processing of a TS_REQUEST message or a TS_PROPOSAL message is done unconditionally by its receiver message. When all the responses have been received by the sender (line 21), the field $final\_ts$ of the message is set to $true$. $\Box_{Lemma\ 2}$

Lemma 3 *When a message gets its final timestamp, no newly inserted message can get a smaller or equal timestamp (even intermediate).*

Proof Let us consider a message m in a given queue $queue_i$ of a task T_i after it got its final timestamp. When m got its final timestamp (line 21 or 30) the Lamport clock h_i has been updated to $max(h_i, m.ts)$ (line 22 or 31). From the management of the local clocks (h_i) that are non-decreasing, all messages that will be inserted afterwards in $queue_i$ (line 6 or 15) will have as an initial intermediate timestamp the value of $h_i > m.ts$ due to lines 2 and 11). Due to the way the final timestamp is computed (it is equal to the maximum of all timestamp proposals) it is greater or equal than all intermediate timestamps of the same message. Consequently, the final timestamp of those messages is greater than $m.ts$. $\Box_{Lemma\ 3}$

Proof (Termination) Let us consider a non-aborted message m. By Lemma 2, m eventually gets its final timestamp. Let s be the set of messages ranked before m at that moment. By Lemma 3, no new message will belong to s (s can only decrease). At the latest, the messages in s will be either aborted and consequently removed from the queue thus from s (Lemma 1) or will get their final timestamps (Lemma 2). If the final timestamp is greater than $m.ts$ the message is also no more member of s. Hence, eventually all messages yet in s will have final timestamps. When a message gets its final timestamp and becomes head of the queue of T_i, it is withdrawn from the queue when it is committed if p_i is the sender. If p_i is a destination process, it withdraws it from the queue provided that p_i issues "enough" DELIVER statements. Eventually message m becomes head of the queue and will be removed from all the queues (after its commit or its delivery). $\Box_{Termination}$

4.4. Logical Instantaneity

vspace3mm

Proof (Instantaneity) This property is made of two parts LI_1 and LI_2 (Section 2.3). Note that no two messages sent by the same process can have the same final timestamp (a process can start a new send statement only after completion of the previous send). Moreover two messages sent by different processes have different sender fields. Consequently, all non-aborted messages are totally ordered by a pair composed of the final timestamp of a message and the identity of its sender [13]. LI_2 follows from the fact that the send event and the corresponding delivery events of each non-aborted message receive the same timestamp. LI_1 comes (1) from the fact that non-aborted messages are removed from the queues according to their timestamp order, and (2) from Lemma 3. (1) and (2) imply that the series of pairs (final timestamp, sender identity) of non-aborted messages removed from the different queues are strictly increasing. $\Box_{Instantaneity}$

5. Conclusion

Communication is *Logically Instantaneous* (LI) if there is a time frame in which matching send and delivery events occur at the same date. LI is stronger than Causally Ordered communication, but weaker than Rendezvous communication. This paper has explored *Logically Instantaneous* communication and has provided an efficient protocol that implements it on top of asynchronous distributed systems.

REFERENCES

1. Mostefaoui A., Raynal M. and Veríssimo P., Logically Instantaneous Communication on Top of Distributed Memory Parallel Machines. *In Proc. of the 5th Int. Conf. on Parallel Computing Technologies (PaCT'99)*, Saint-Petersburg, LNCS 1662, pp. 258-270, september 1999.
2. Powell D. (Guest Editor), Special Issue on Group Communication. *CACM*, 39(4), (1996).
3. Birman K.P. and Joseph T.A., Reliable Communication in the Presence of Failures. *ACM TOCS*, 5(1):47-76, (1987).
4. 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).
5. Garcia-Molina H. and Spauster A., Ordered and Reliable Multicast Communication. *ACM TOCS*, 9(3):242-272, (1991).
6. Hadzilacos V. and Toueg S., Reliable Broadcast and Related Problems. In *Distributed Systems*, ACM Press (S. Mullender Ed.), New-York, pp. 97-145, (1993).

7. Bagrodia R., Process Synchronization: Design and Performance Evaluation for Distributed Algorithms. *IEEE TSE*, SE15(9):1053-1065, (1989).
8. Bagrodia R., Synchronization of Asynchronous Processes in CSP. *ACM TOPLAS*, 11(4):585-597, (1989).
9. Hoare C.A.R., Communicating Sequentail Processes. *Communications of the ACM*, 21(8):666-677, (1978).
10. Bougé L., Repeated Snapshots in Distributed Systems with Synchronous Communications and their Implementation in CSP. *Theoretical Computer Science*, 49:145-169, (1987).
11. Charron-Bost B., Mattern F. and Tel G., Synchronous, Asynchronous and Causally Ordered Communications. *Distributed Computing*, 9:173-191, (1996).
12. Soneoka T. and Ibaraki T., Logically Instantaneous Message Passing in Asynchronous Distributed Systems. *IEEE TC*, 43(5):513-527, (1994).
13. Lamport, L., Time, Clocks and the Ordering of Events in a Distributed System, *Communications of the ACM*, 21(7):558-565, (1978).
14. Murty V.V. and Garg V.K., Synchronous Message Passing. *Tech. Report* TR ECE-PDS-93-01, University of Texas at Austin, (1993).
15. Silberschatz A., Synchronization and Communication in Distributed Systems. *IEEE TSE*, SE5(6):542-546, (1979).
16. Brinch Hansen P., Distributed Processes: A Concurrent Programming Concept. *Communications of the ACM*, 21(11):934-941, (1978).