Logically Instantaneous Communication on top of Distributed Memory Parallel Machines

Achour MOSTEFAOUI Michel RAYNAL
IRISA, Campus de Beaulieu
35 042 Rennes-cedex, France
{mostefaoui,raynal}@irisa.fr

Paulo VERÍSSIMO
DI-FCUL, C5 Campo Grande
1700 Lisboa, Portugal
pjv@di.fcul.pt

Abstract

Communication is *Logically Instantaneous* (LI) if 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.

LI is stronger than Causally Ordered (CO) communication, but weaker than Rendezvous (RDV) communication. LI is attractive as it includes CO and provides more concurrency than RDV. Moreover it allows to adopt the following approach: first design a distributed application assuming Rendezvous communication, and then run it on top of an asynchronous distributed system providing only LI 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.

1 Introduction

Among the services offered by a distributed 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 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, 18] and *total order* (TO) [4, 7, 11] are the most encountered ordering properties [12]. 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 [9]. The ISIS system [4] pioneered research in this domain. Other systems that have proposed powerful communication services are described in [17].

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 [2, 3, 13] (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" [8]. 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 [21]). 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 [14] and concern the same destination process, then the corresponding messages are delivered in their sending order [4]. CO has received a great attention as it simplifies the design of protocols solving consistency-related problems [18]. It has been shown that these communication modes form a strict hierarchy [8, 21]. 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. 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[1]: first design (and prove) programs[2] assuming RDV communications; and then run the obtained programs on top of a system offering LI communications. 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 four sections. Section 2 introduces the underlying asynchronous distributed system model and defines logically instanta-

[1] This is similar to the *Distributed Shared Memory* (DSM) paradigm. DSM allows to execute parallel programs, made of processes communicating through shared variables, on top of distributed memory systems.

[2] We only consider here non realtime programs.

neous communication. Then, Section 3 presents a protocol implementing LI communication on top of an asynchronous distributed system. Finally, Section 4 considers the case of unreliable systems and 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 $\{P_1, \ldots, P_n\}$ 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[3]. 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, *i.e.*, they do not suffer failures (Section 4 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 [14]:

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

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 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$_j(m)$.

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

Previous proposals [16, 21] 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[4].
- *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. It exists a timestamping function $\mathcal{T}$ from H_a into $I\!N$ such that $\forall (e, f) \in H_a \times H_a$ [16]:

$$\begin{array}{lll} (LI_1) & e \text{ and } f \text{ have been produced by the same process with } e \text{ first} & \Rightarrow \mathcal{T}(e) < \mathcal{T}(f) \\ (LI_2) & \forall m : \forall j \in dest_m : e = \text{SEND}_{m.sender}(m) \wedge f = \text{DELIVER}_j(m) & \Rightarrow \mathcal{T}(e) = \mathcal{T}(f) \end{array}$$

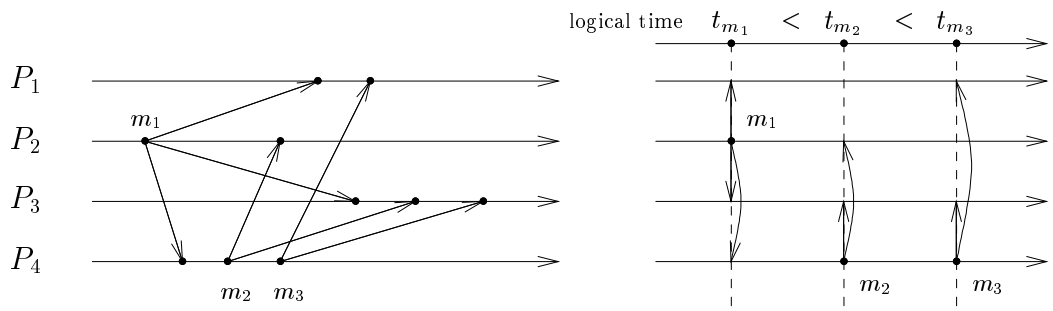
From the point of view of the communication of a message m, the event SEND$_{m.sender}(m)$ is the cause and the events DELIVER$_j(m)$ $(j \in dest_m)$ are the effects. The termination property associates effects with a cause. The validity property associates a cause with each effect (in other words, 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.

[4]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.



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

Figure 1: Logical Instantaneity

- *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 [20] (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. There is no way to a priori know which primitive will be 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 [14]. 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 [4].)

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.

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[5]. 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 described in Figure 2, 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 ∧ head(queue_i).final_ts;
    commit(msg_id); /* Event SEND_i(m) */
    remove head(queue_i) from queue_i
```

Figure 2: Deterministic Context: SEND Protocol

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 $queue_i$.

[5]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 [14]).

DELIVER **in a deterministic context**

DELIVER in a deterministic context is implemented by the procedure described in Figure 3. 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 ← m.content; /* Event DELIVER_i(m) */
    remove m from queue_i
```

Figure 3: Deterministic Context: DELIVER Protocol

SEND/DELIVER **in a non-deterministic context**

This non-deterministic context communication statement is implemented by the function described in Figure 4. First P_i initiates the sending of the message (so, the first two lines are the same as in Figure 2). 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   →  commit(msg_id); /* Event SEND_i(m) */
                            return "sending"
         m.id ≠ msg_id   →  abort(msg_id);
                            del_msg ← m.content; /* Event DELIVER_i(m) */
                            return "delivery"
    endcase;
    remove m from queue_i
```

Figure 4: Non-Deterministic Context: SEND/DELIVER Protocol

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 [6]:

- 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 [14]. The other two, namely, $dest_i$ and $wait_ts_from_i$, are relevant only when P_i is engaged in a SEND(m) primitive (Figure 2 or Figure 4). 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 (Figure 2 or Figure 4), 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 (see Figure 2, Figure 3 or Figure 4), or its sending is aborted by its sender (see Figure 4) ([6]).

- If the sender process commits the sending of m (Figure 2 or Figure 4), 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 (Figure 4), 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.

3.4 Sketch of Proof

Due to space limitation, only a proof sketch is given.

Integrity and Validity
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.

[6] Actually, this part of the protocol is close to Ricart-Agrawala's mutual exclusion protocol [19]. Both use the same message exchange pattern: one for ordering the critical section requests, the other for ordering the messages whose sending have been committed.

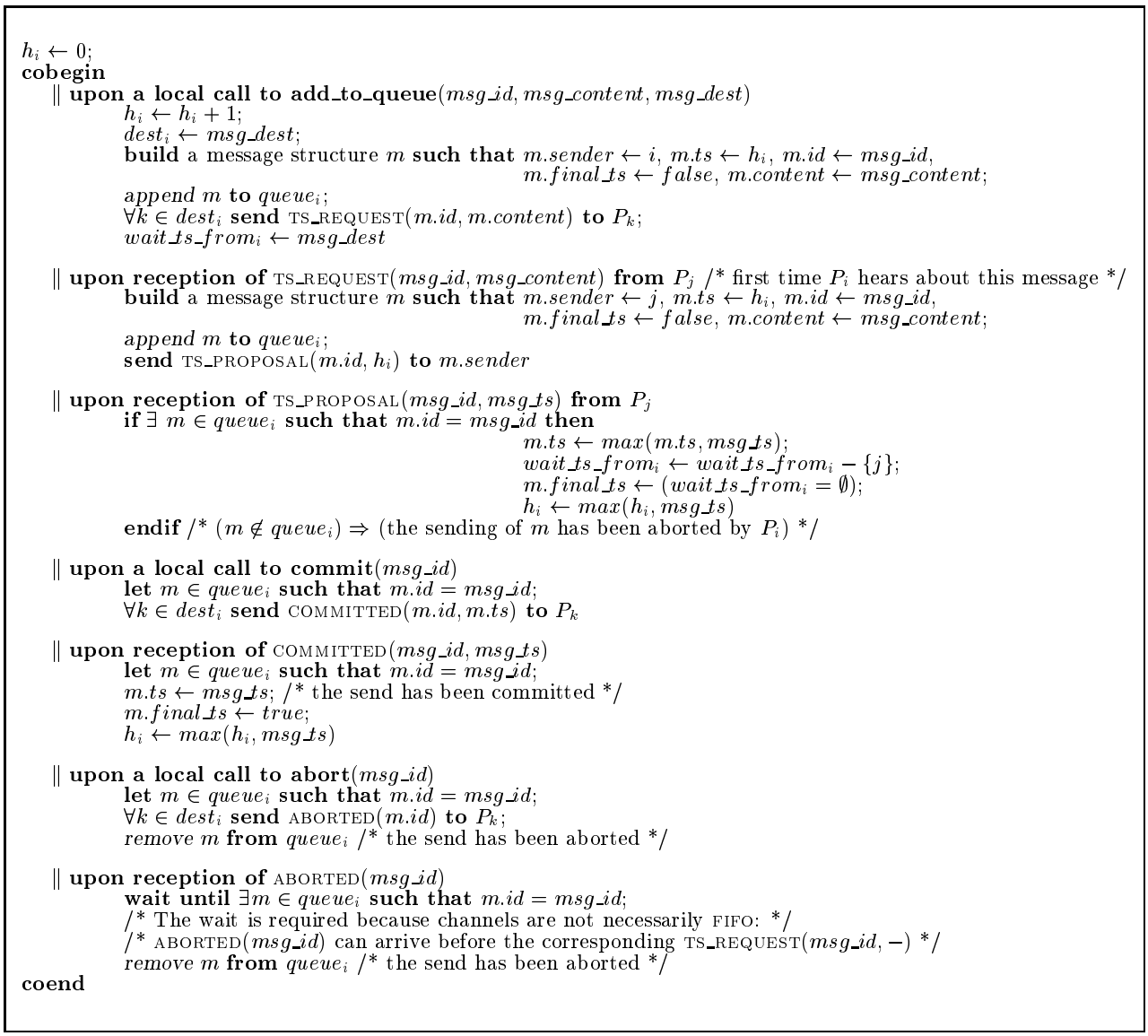


Figure 5: The Communication Task T_i

The validity property (if a message is LI-delivered, it has been LI-sent) follows from 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 $true$ this message can not be delivered.

Logical Instantaneity
This property is made of two parts LI_1 and LI_2 (Section 2.3). Note that all non-aborted messages are totally ordered by their timestamps [14]. 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 management of the local clocks (h_i) that are non-decreasing, and (2) from the fact that non-aborted messages are removed from the queues according to their timestamp order.

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

Let us first remark that each message that has been aborted is eventually withdrawn from the queues in which it has been added. So, it is not necessary to consider aborted messages: they cannot prevent committed messages from arriving at the head of the queue. So, in the following we consider only committed messages.

The proof follows from the fact that the two-way handshake sub-protocol that computes the final timestamp of m (implemented by the sequence of control messages TS_REQUEST, TS_PROPOSAL) is not blocking. The processing of a TS_REQUEST or TS_PROPOSAL message is done unconditionally by its receiver and any TS_REQUEST entails the sending of the corresponding TS_PROPOSAL.

So, there is eventually a message m at the head of all $queue_j$ ($\forall j \in dest_m \cup \{m.sender\}$) such that $m.final_ts = true$. This message can be delivered by all destination processes. The sender withdraws it from its queue when it commits the sending; a destination process withdraws it from its queue when it LI-delivers it. Then a simple induction shows all committed messages can be LI-delivered (provided that their destination processes issue "enough" DELIVER).

3.5 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 [14] and has the same message cost, namely $3(n-1)$.

4 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.

LI communication is very close to what is needed to implement a parallel discrete event simulation system. In a discrete event simulation model, actions issued by agents consume time but their interactions do not [10, 15]. The main problem of a simulation model is to implement the so called *virtual time* (simulation time). Among other uses, the proposed LI protocol can be used to implement agent interactions in such a system.

Last but not least, let us note that the proposed protocol can be extended to cope with process crashes by using the techniques introduced and used 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., Process Synchronization: Design and Performance Evaluation for Distributed Algorithms. *IEEE TSE*, SE15(9):1053-1065, (1989).

[3] Bagrodia R., Synchronization of Asynchronous Processes in CSP. *ACM TOPLAS*, 11(4):585-597, (1989).

[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. *Theoretical Computer Science*, 49:145-169, (1987).

[6] Brinch Hansen P., Distributed Processes: A Concurrent Programming Concept. *Communications of the ACM*, 21(11):934-941, (1978).

[7] Chandra T. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(1):225–267, (1996).

[8] Charron-Bost B., Mattern F. and Tel G., Synchronous, Asynchronous and Causally Ordered Communications. *Distributed Computing*, 9:173-191, (1996).

[9] Fritkze U., Ingels Ph., Mostefaoui, A. and Raynal M., Fault-Tolerant Total Order Multicast to Asynchronous Groups. *Proc. 16th IEEE Int. Symposium on Reliable Distributed Systems*, Purdue University, pp. 228-234, (1998).

[10] Fujimoto R., Parallel Discrete Event Simulation. *Communications of the ACM*, 33(10):31-53, (1990).

[11] Garcia-Molina H. and Spauster A., Ordered and Reliable Multicast Communication. *ACM TOCS*, 9(3):242-272, (1991).

[12] Hadzilacos V. and Toueg S., Reliable Broadcast and Related Problems. In *Distributed Systems*, ACM Press (S. Mullender Ed.), New-York, pp. 97-145, (1993).

[13] Hoare C.A.R., Communicating Sequentail Processes. *Communications of the ACM*, 21(8):666-677, (1978).

[14] Lamport, L., Time, Clocks and the Ordering of Events in a Distributed System, *Communications of the ACM*, 21(7):558-565, (1978).

[15] Misra J., Distributed Discrete Event Simulation. *ACM Computing Surveys*, 18(1):39-65, (1986).

[16] Murty V.V. and Garg V.K., Synchronous Message Passing. *Tech. Report* TR ECE-PDS-93-01, University of Texas at Austin, (1993).

[17] Powell D. (Guest Editor), Special Issue on Group Communication. *CACM*, 39(4), (1996).

[18] 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).

[19] Ricart G. and Agrawala A.K., An Optimal Algorithm for Mutual Exclusion in Computer Networks. *Communications of the ACM*, 24(1):115-123, (1981).

[20] Silberschatz A., Synchronization and Communication in Distributed Systems. *IEEE TSE*, SE5(6):542-546, (1979).

[21] Soneoka T. and Ibaraki T., Logically Instantaneous Message Passing in Asynchronous Distributed Systems. *IEEE TC*, 43(5):513-527, (1994).