



**HAL**  
open science

## Send/Receive Patterns versus Read/Write Patterns: the MB-Broadcast Abstraction (Extended Version)

Mathilde Déprés, Achour Mostefaoui, Matthieu Perrin, Michel Raynal

### ► To cite this version:

Mathilde Déprés, Achour Mostefaoui, Matthieu Perrin, Michel Raynal. Send/Receive Patterns versus Read/Write Patterns: the MB-Broadcast Abstraction (Extended Version). 2023. hal-04087447

**HAL Id: hal-04087447**

**<https://hal.science/hal-04087447>**

Preprint submitted on 3 May 2023

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Send/Receive Patterns versus Read/Write Patterns: the MB-Broadcast Abstraction (Extended Version)

Mathilde Déprés<sup>†</sup>, Achour Mostéfaoui<sup>\*</sup> Matthieu Perrin<sup>\*</sup>, Michel Raynal<sup>◇</sup>

<sup>†</sup>ENS Paris-Saclay, Université Paris-Saclay, 91190 Gif-sur-Yvette, France

<sup>\*</sup>LS2N, Nantes Université, 44322 Nantes, France

<sup>◇</sup>Univ Rennes, IRISA, Inria, CNRS, 35042 Rennes, France

## Abstract

This paper is on the power and computability limits of messages patterns in crash-prone asynchronous message-passing systems. It proposes and investigates three basic messages patterns (encountered in all these systems) each involving two processes, and compares them to their Read/Write counterparts. It is first shown that one of these patterns has no Read/Write counterpart.

The paper proposes then a new one-to-all broadcast abstraction, denoted *Mutual Broadcast* (in short MBroadcast), whose implementation relies on two of the previous messages patterns. This abstraction provides each pair of processes with the following property (called *mutual ordering*): for any pair of processes  $p$  and  $p'$ , if  $p$  broadcasts a message  $m$  and  $p'$  broadcasts a message  $m'$ , it is not possible for  $p$  to deliver first (its message)  $m$  and then  $m'$  while  $p'$  delivers first (its message)  $m'$  and then  $m$ . It is shown that MBroadcast and atomic Read/Write registers have the same computability power (independently of the number of crashes).

Finally, in addition to its theoretical contribution, the practical interest of MBroadcast is illustrated by its (very simple) use to solve basic upper level coordination problems such as mutual exclusion and consensus. Last but not least, looking for simplicity was also a target of this article.

**Keywords:** Asynchrony, Atomicity, Broadcast abstraction, Characterization, Consensus, Crash failure, Distributed software engineering, Lattice agreement, Message-passing, Message pattern, Mutual exclusion, Quorum, Read/write pattern, Read/Write register, Test&Set, Simplicity, Two-process communication.

## 1 Introduction

### 1.1 On the nature of distributed computing

The aim of parallel computing is to allow programmers to exploit data independence in order to obtain efficient programs. In distributed computing, the situation is different: there is a set of predefined computing entities (imposed to programmers) that need to cooperate to a common goal. Moreover, the behavior of the underlying infrastructure (environment) on which the distributed application is executed is not on the control of the programmers, who have to consider it as a “hidden input”. Asynchrony and failures are the most frequent phenomenons produced by the environment that create a “context uncertainty” distributed computing has to cope with. In short, distributed computing is characterized by the fact that, in any distributed run, the run itself is one of its entries [45].

## 1.2 From send/receive to cooperation abstractions

The operations `send()` and `receive()` constitute the machine language of underlying networks. So, in order to solve a distributed computing problem it is usual to first define an appropriate communication abstraction that makes easier the design of higher level algorithms. FIFO and Causal message-ordering [10, 24, 39, 48] are examples of such communication abstractions that make easier the construction of distributed objects such as, for example, the construction of a causal memory on top of an asynchronous message-passing system [4]. A well-know high level and very powerful communication abstraction is total order broadcast, which ensures that the message delivery order is the same at all processes.

Another example is the *Set-Constrained Delivery* (SCD) communication abstraction introduced in [30]. This broadcast abstraction allows processes to deliver a sequence of sets of messages of arbitrary size (instead of a sequence of messages) satisfying a non trivial intersection property. In terms of computability in the presence of asynchrony and process crashes, the power of SCD-broadcast is the same as the one of atomic read/write registers. SCD-broadcast is particularly well-suited to efficiently implement a snapshot object (as defined in [1, 5]) with an  $O(n^2)$  message complexity in asynchronous crash-prone message-passing systems. Broadcast abstractions suited to specific problems have also been designed (e.g., [29] for  $k$ -set agreement).

$d$ -Solo models consider asynchronous distributed systems where any number of processes may crash. In these models, up to  $d$  ( $1 \leq d \leq n$ ) processes may have to run solo, computing their local output without receiving any information from other processes. Differently from the message-passing communication model where up to  $d \geq 1$  processes are allowed to run solo, it is important noticing that the basic atomic read/write registers communication model allows at most one process to run solo. Considering the family of  $d$ -solo models, [26] presents a characterization of the colorless tasks that can be solved in each  $d$ -solo model.

## 1.3 On the read/write side

Read/write (RW) registers (i.e., the cells of a Turing machine) are at the center of distributed algorithms when the processes communicate through a shared memory. So a fundamental problem is the construction of atomic RW registers on top of crash-prone asynchronous message-passing system. This problem has been solved by Attiya, Bar-Noy and Dolev who presented in [7] a send/receive-based algorithm (ABD) for such a construction, and proved that, from an operational point of view, such constructions are possible if and only if at most  $t < n/2$  processes may crash. The ABD construction is based on the explicit use of sequence numbers, quorums and a send/receive pattern (used once to write and twice to read). The quorums are used to realize synchronization barriers<sup>1</sup>.

Based on the same principles as ABD, algorithms building RW registers have been designed [50]. Some strive to reduce the size of control information carried by messages (e.g. [6,40]) while others focus on fast read and/or fast write operations in good circumstances (e.g. [18, 23, 41]). All these algorithms allow existing RW-based algorithms to be used on top of crash-prone asynchronous message-passing systems.

The use of RW registers on top of a message-passing system to allow processes to use “for free” existing shared memory-based distributed algorithms has a cost, which can be higher than the one obtained with an algorithm directly designed on top of the basic send/receive operations (as shown, for example, in [15] for the snapshot object defined in [1, 5]). This means that, for some problems (as we will see for consensus), algorithms based on appropriate communication abstractions can be more efficient than the stacking of RW-based algorithms on top of simulated RW registers.

---

<sup>1</sup>Let us notice that, in the traditional use of the send/receive patterns, a process that broadcasts a message (i.e., sends a message  $m$  to all the processes including itself) is allowed to locally deliver  $m$  without waiting for a specific delivery condition to be satisfied.

## 1.4 Content of the article

In the spirit of [21] (which states that computing is “science of abstractions”) the present article introduces a new broadcast abstraction (denoted MBroadcast) that ensures that for any pair of processes  $p$  and  $p'$ , if  $p$  broadcasts a message  $m$  and  $p'$  broadcasts a message  $m'$ , it is not possible for  $p$  to deliver first its message  $m$  and then  $m'$  while  $p'$  delivers first its message  $m'$  and then  $m$ . It is important to notice that this property is on each pair of processes taken separately from the other processes. It is also shown how, at the upper layer, this broadcast abstraction allows a very simple design of message-passing algorithms solving distributed coordination and agreement problems. The main properties of MBroadcast are the following ones.

- It has the same computability power as RW registers.
- It constitutes the first characterization of RW registers in terms of (binary) message patterns.
- It allows to build higher level coordination abstractions without requiring as prerequisite the construction of an intermediary abstraction level made up of RW registers.
- Its cost is  $O(n)$  implementation messages when mutual broadcast delivery concerns only correct processes.
- It makes quorums useless at the upper layer.
- When looking at a message exchange between two processes  $p$  and  $p'$ , it shows that the fact that  $p'$  does not ignore the other process  $p$  (because it received a message from  $p$  before receiving its own message) is a powerful control information (more technically, this refers to the patterns MP2 and MP3 defined in Section 3).

An important point of the article is the fact that atomic RW registers can be implemented from two simple basic message patterns on each pair of processes only. Hence the article is on basic patterns that allow us to better understand the close relationship between RW registers and asynchronous message-passing in the presence of process crashes. More generally, it allows for a better understanding of the strengths and weaknesses of the world of asynchronous crash-prone message-passing systems.

As an important side note, this article also discusses a strengthening of MBroadcast, denoted PBroadcast, that ensures that any pair of processes deliver their own messages in the same order (in the terms of the patterns defined in Section 3, it means that the only possible pattern is MP2). On a computability viewpoint, it shows that if the only binary message pattern that can occur is MP2, then Test&Set, the consensus number of which is 2, can be built.

## 1.5 Roadmap

The article is composed of 8 sections, structured in two parts. The first part consists of sections 2-5, while the second part consists of sections 6-7. Section 2 presents the underlying computing model. Section 3 presents three basic binary message patterns (denoted MP1, MP2 and MP3) encountered in asynchronous message-passing computations, and establishes a “correspondence” linking these message patterns with read/write patterns on atomic registers. Section 4 introduces the high level MBroadcast and PBroadcast communication abstractions. Section 5 shows that MBroadcast and atomic read/write (RW) registers have the same computability power.

The second part illustrates uses of MBroadcast, that show the conceptual gain offered by this communication abstraction (i.e., simplicity). More precisely, Section 6 presents an MBroadcast-based

rewriting of Lamport’s bakery algorithm suited to message-passing (i.e., suited to state machine replication [35]). Section 7 presents a simple version of the well-known Paxos consensus algorithm [34].<sup>2</sup> It is important to state that none of the algorithms built on top of MBroadcast uses quorums (as we will see, this means if each binary message exchange pattern satisfies the pattern MP2 or the pattern MP3 these algorithms work correctly even if a majority of processes crashes).

Finally, Section 8 concludes the article (where design simplicity is considered as a first class citizen property).

## 2 Distributed Computing Model

The computing model is the classical asynchronous crash-prone message-passing model.

### 2.1 Process model

The computing model is composed of a set of  $n$  sequential processes denoted  $p_1, \dots, p_n$ . Sometimes, when considering two processes, they are denoted  $p$  and  $p'$ .

Each process is asynchronous which means that it proceeds at its own speed, which can be arbitrary and remains always unknown to the other processes. A process may halt prematurely (crash failure), but executes correctly its local algorithm until it possibly crashes. The model parameter  $t$  denotes the maximal number of processes that may crash in a run. A process that crashes in a run is said to be *faulty*. Otherwise, it is *correct* or *non-faulty*.

### 2.2 Communication model

Each pair of processes communicate by sending and receiving messages through two uni-directional channels, one in each direction. Hence, the communication network is a complete network: any process  $p_i$  can directly send a message to any process  $p_j$  (including itself). A process  $p_i$  invokes the operation “send TYPE( $m$ ) to  $p_j$ ” to send the message  $m$  (whose type is TYPE) to  $p_j$ . The operation “receive TYPE( $m$ ) from  $p_j$ ” allows  $p_i$  to receive from  $p_j$  a message  $m$  whose type is TYPE.

Each channel is reliable (no loss, no corruption, no creation of messages), not necessarily first-in/first-out, and asynchronous (while the transit time of each message is finite, there is no upper bound on message transit times). Let us notice that, due to process and message asynchrony, no process can know if another process crashed or is only very slow.

It is assumed that, in addition to basic send/receive operations, the network is enriched with FIFO and causal message deliveries, i.e. it provides the processes with the operations `fifo_broadcast()`, `causal_broadcast()`, `causal_send()`, and `causal_delivery()` [10, 24, 32, 39, 45, 48]. Two messages  $m$  and  $m'$  are causally related if (i) they have been sent by the same process and  $m$  was sent before  $m'$ , or (ii) a process received  $m$  before sending  $m'$ , or (iii) there is chain of messages  $m, m_1, \dots, m_x, m'$  such that each pair of consecutive messages is causally related by (i) or (ii). It is important to note that the addition of this assumption does not change the computability power of the communication model, since causal message delivery can always be implemented on top of send/receive channels. It is also important to note that, while the implementation of these operations requires messages to carry additional control information, it does not require the use of additional implementation messages.

---

<sup>2</sup>It is worth noticing that mutex and consensus are the two most famous distributed computing problems [47]. Additionally, Appendix B presents a very simple MBroadcast-based algorithm that builds a lattice agreement object.

### 2.3 Notation

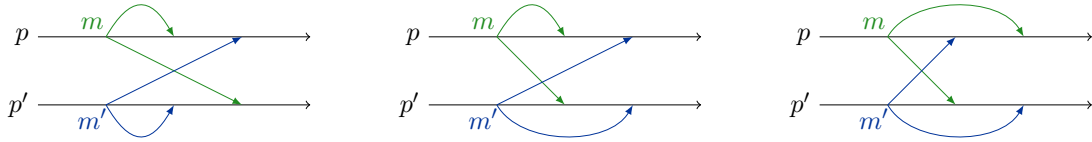
The acronym  $\mathcal{CAMP}_{n,t}[\emptyset]$  is used to denote the previous Crash-prone Asynchronous Message-Passing model without additional computability power.  $\mathcal{CAMP}_{n,t}[H]$  denotes  $\mathcal{CAMP}_{n,t}[\emptyset]$  enriched with the additional computational power denoted by  $H$ .

The acronym  $\mathcal{CARW}_{n,t}[\emptyset]$  is used to denote the  $n$ -process asynchronous system where up to  $t$  processes may crash and communication is through read/write registers.  $\mathcal{CARW}_{n,t}[H]$  denotes  $\mathcal{CARW}_{n,t}[\emptyset]$  enriched with  $H$ .

## 3 Three Basic Binary Message Patterns and their RW Counterparts

### 3.1 Three basic binary message patterns

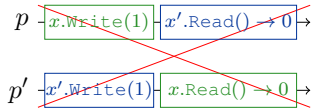
Let us consider two processes  $p$  and  $p'$  that concurrently exchange messages, namely,  $p$  sends a message  $m$  to itself and  $p'$ , while  $p'$  sends the message  $m' \neq m$  to itself and  $p$ . Depending on the order in which messages are delivered at each process, there are exactly three cases to consider (swapping  $p$  and  $p'$  does not give rise to new message patterns).



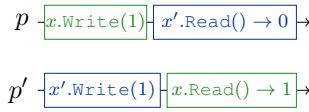
(a) Pattern MP1:  
forbidden by MBroadcast  
and by PBroadcast

(b) Pattern MP2:  
allowed by MBroadcast  
and by PBroadcast

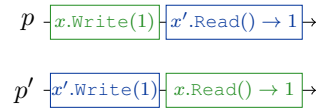
(c) Pattern MP3:  
allowed by MBroadcast,  
forbidden by PBroadcast



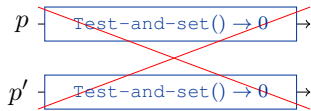
(d) Pattern RW1:  
forbidden by atomic memory



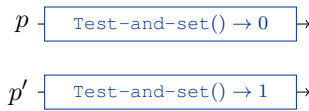
(e) Pattern RW2:  
allowed by atomic memory



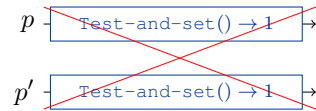
(f) Pattern RW3:  
allowed by atomic memory



(g) Pattern TS1:  
forbidden by atomic Test&Set



(h) Pattern TS2:  
allowed by atomic Test&Set



(i) Pattern TS3:  
allowed by atomic Test&Set

Figure 1: The three binary message patterns, versus the three binary atomic memory patterns and the three binary atomic test-and-set patterns

**Message pattern MP1.** This case is represented at Figure 1a. It is a symmetric pattern in which  $p$  delivers first the message  $m$  it broadcast and then the message  $m'$  broadcast by  $p'$ , while  $p'$  delivers first its message  $m'$  and then the message  $m$  broadcast by  $p$ . This pattern captures the case where, when a process delivers its own message, it has no information on the fact the other processes broadcast or not a message.

**Message pattern MP2.** This case is represented at Figure 1b. It describes an asymmetric pattern from a message delivery point of view in which both  $p$  and  $p'$  deliver first  $m$  broadcast by  $p$  and then  $m'$

broadcast by  $p'$ . In this pattern both  $p$  and  $p'$  deliver the messages in the same order (an analogous pattern occurs when we swap  $p$  and  $p'$ ).

**Message pattern MP3.** This case is represented at Figure 1c. Similarly to MP1 this is a symmetric pattern in the sense that  $p$  delivers first the message  $m'$  from  $p'$  and then its message  $m$ , while  $p'$  delivers first the message  $m$  from  $p$  and then its message  $m'$ . The fundamental difference between MP1 and MP3 lies in the fact that when  $p$  (resp.  $p'$ ) delivers its own message, it has already delivered the message sent by the other process  $p'$  (resp.  $p$ ).

### 3.2 From message patterns to RW patterns

To deeply understand the meaning and the scope of the three previous message-based communication patterns, let us consider their “counterpart” in a context where  $p$  and  $p'$  cooperate through atomic one-bit RW registers initialized to 0. The RW register  $x$ , written by  $p$  and read by  $p'$ , corresponds to  $m$ . The RW register  $x'$ , written by  $p'$  and read by  $p$  corresponds to  $m'$ . Both registers are initialized to 0. In each case, the read/write pattern depicted at the bottom of Figure 1 simulates the message exchange pattern above it. More precisely we have the following.

**RW pattern RW1.** Figure 1d corresponds to the message pattern MP1:  $p$  writes 1 in  $x$  and reads the initial value of  $x'$ , namely, the value 0. Concurrently,  $p'$  writes 1 in  $x'$  and reads the initial value of  $x$ , namely, the value 0.

**RW pattern RW2.** Figure 1e corresponds to the message pattern MP2:  $p$  writes 1 in  $x$  and then reads the initial value 0 from  $x'$ , while  $p'$  writes 1 in  $x'$  and then reads the value of  $x$ , namely, the value 1.

**RW pattern RW3.** Figure 1f corresponds to the message pattern MP3: each process writes first “its” variable ( $x$  for  $p$ ,  $x'$  for  $p'$ ), and then reads the other variable and obtains 1.

### 3.3 Comparing message patterns and RW patterns

It is easy to see that the RW patterns RW2 and RW3 produce the same cooperation as the message patterns MP2 and MP3, respectively. Differently, while the message pattern MP1 can occur in an asynchronous message-passing system, the RW pattern RW1 cannot occur in a RW memory. This is due to the fact that, in RW1, the write of  $x$  by  $p$  and the write of  $x'$  by  $p'$  are linearized [27, 33] and, as a process writes a RW register before reading the other register, it is impossible that both read operations return 0 (it is easy to show that this remains true if the registers are only safe, regular, or part of a sequentially consistent memory). This is a fundamental difference between cooperation/communication through message passing and cooperation/communication through RW registers.

### 3.4 On the test-and-set side

For comparison, Figures 1g, 1h and 1i consider three communication patterns where the two processes cooperate through the **Test&Set** special instruction on an atomic register. In a similar way as with message patterns and RW patterns, we can define three TS patterns, depending on which processes obtain 0, the same outcome as in a solo execution; and which processes obtain the same outcome 1 as if their operation was linearized in second position.

**TS pattern TS1.** Figure 1g corresponds to the RW pattern RW1, in which both processes obtain 0, and hence to the message pattern MP1.

**TS pattern TS2.** Figure 1h corresponds to the asymmetric RW pattern RW2, in which one process obtains 0 and the other process obtains 1. Hence, it also corresponds to the message pattern MP2.

**TS pattern TS3.** Figure 1i corresponds to the symmetric RW pattern RW3, in which both processes obtain 1, and hence to the message pattern MP3.

This time, only the asymmetric communication pattern TS2 is admitted. The fact that the pattern TS3 is impossible is a major difference between read/write registers and test-and-set registers, that can be used to solve consensus between two processes.

### 3.5 At the core of the approach

The fact that there is no RW pattern corresponding to pattern MP1 is implicitly used to solve many cooperation/synchronization problems in RW systems. The most famous is the “write first and then read” pattern used in all mutex algorithms [49]. When a process wants to enter the critical section, it first raises a flag to inform the other processes it starts competing, and only then it reads the flags (which are up or down) of the other processes. The total order imposed by the atomicity on the flag risings prevents RW1 from occurring.

Actually preventing the message pattern MP1 from occurring without bounding the number of process crashes is “equivalent” to the assumption  $t < n/2$  without constraints on message exchange patterns, in the sense that both prevent partitioning and consequently allow atomic RW registers to be built despite crashes and asynchrony.

## 4 Mutual Broadcast

### 4.1 Mutual broadcast: Definition

Mutual-broadcast (MBroadcast) is a broadcast abstraction that allows a process to broadcast a message that will be delivered at least by all the correct processes. This abstraction provides the processes with two operations denoted `mb_broadcast()` and `mb_deliver()`. When a process invokes `mb_broadcast( $m$ )` we say “it mbroadcasts the message  $m$ ”. The invocation of `mb_deliver()` returns a message  $m$  and we say that a process “mdelivers  $m$ ” or that “ $m$  is mdelivered”. To simplify the presentation (and without loss of generality), it is assumed that all the messages that are mbroadcast are different. The following properties define MBroadcast.

**Validity.** If a process  $p_i$  mdelivers a message  $m$  from a process  $p_j$ , then  $p_j$  previously invoked `mb_broadcast( $m$ )`.

**No-duplication.** A process mdelivers a message  $m$  at most once.

**Mutual ordering.** For any pair of processes  $p$  and  $p'$ , if  $p$  mbroadcasts a message  $m$  and  $p'$  mbroadcasts a message  $m'$ , it is not possible that  $p$  mdelivers  $m$  before  $m'$  and  $p'$  mdelivers  $m'$  before  $m$ .

**Local termination.** If a correct process invokes `mb_broadcast( $m$ )`, it returns from its invocation.

**Global CS-termination.** If a correct process invokes `mb_broadcast( $m$ )`, all correct processes mdeliver  $m$ . (“CS” is used to stress the fact that the sender is required to be correct.)

Let us notice that, at the user level, the Mutual ordering property prevents the pattern MP1 from occurring, boils down to the pattern MP2 when a process delivers its message first, and does not prevent pattern MP3 from occurring (which occurs when each process delivers first the message from the other process before its own message).



**Additional properties** As it is the case with other broadcast abstractions, MBroadcast can be enriched with other properties, defined thereafter, that do not change its computing power but add more usage comfort in some algorithms. We denote by *fifo-MBroadcast*, *causal-MBroadcast* and *reliable-MBroadcast* the MBroadcast abstraction respectively enriched with the following properties:

**FIFO ordering.** If a process *mbroadcasts* a message  $m$  and then *mbroadcasts* a message  $m'$ , no process *mdelivers*  $m'$  before  $m$ .

**Causal ordering.** If a process *mdelivers* a message  $m$  and then *mbroadcasts* a message  $m'$ , no process *mdelivers*  $m'$  before  $m$ .

**Global CF-termination.** If a process *mdelivers* a message  $m$ , all correct processes *mdeliver*  $m$ . (“CF” is used to stress the fact that the receiver of  $m$  can be correct or faulty.)

**Remark on reliable broadcast** It is easy to see that the classical reliable broadcast communication abstraction is MBroadcast with the Global CF-termination property and without the Mutual ordering property. Similarly, the causal broadcast abstraction (denoted *causal\_broadcast* in the following) is MBroadcast with the Causal ordering and Global CF-termination properties and without the Mutual ordering property.

## 4.2 What does MBroadcast do

When looking at MBroadcast from a *binary* communication point of view between two processes  $p$  and  $p'$ , it appears that MBroadcast ensures that, for any message exchanged by  $p$  and  $p'$ , the message patterns produced is MP2 or MP3. Moreover, it is possible that some of their message exchange patterns are MP2 while others are MP3, and this remains unknown to the processes.

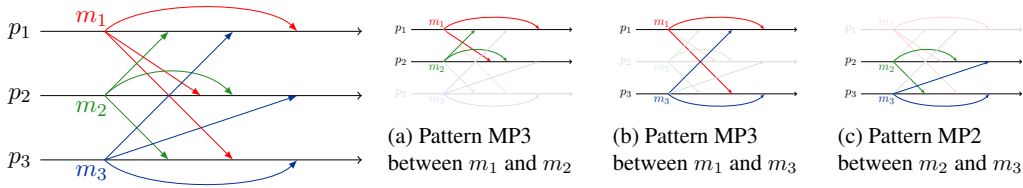


Figure 2: Example of message deliveries with three processes

Figure 2 presents an example with three processes, that shows MBroadcast message deliveries. As we can see  $p_1$  *mdelivers* the sequence of messages  $m_2, m_3, m_1$ ,  $p_2$  *mdelivers* the sequence of messages  $m_1, m_2, m_3$ , and  $p_3$  *mdelivers* the sequence of messages  $m_2, m_1, m_3$ . So, the processes *mdeliver* the three messages in different orders. Nevertheless, when we consider the projection of these messages exchange on each pair of processes, we observe that the processes  $p_1$  and  $p_2$  *mdeliver*  $m_1$  and  $m_2$  according to the pattern MP3. As depicted in Figure 2.a each process *mdelivers* the message from the other process before its own message. The same message pattern MP3 occurs for the messages exchanged by  $p_1$  and  $p_3$  (Figure 2.b). Differently, as shown in Figure 2.c, the messages exchanged by  $p_2$  and  $p_3$  obey the pattern MP2: both processes *mdeliver* first  $m_2$  and then  $m_3$ . The fundamental point is that the pattern MP1 never occurs: MBroadcast prevents it from occurring, which implicitly prevents system partitioning and makes the system parameter  $t$  useless in the algorithms build on top of MBroadcast.

## 4.3 A real-time property

When combined with the fact that a message cannot be received before it has been sent, MBroadcast ensures that if a process  $p$  *mdelivers* a message  $m$  it has *mbroadcast* before a process  $p'$  *mbroadcast* a

message  $m'$ ,  $p'$  cannot deliver  $m'$  before  $m$ . This pattern is described in Figure ?? . In other words, sending a “blank” synchronization message and waiting for its delivery is sufficient to “harvest” all the messages that were already delivered by their senders. To benefit from this property, we consider the synchronized broadcast operation based on a synchronization barrier as defined in Algorithm 1.

```

operation synchro_mbroadcast( $m$ ) is      % code for  $p_i$ 
(1)  mb_broadcast( $m$ );
(2)  wait ( $m$  has been mb-delivered from  $p_i$ )
end operation.

```

Algorithm 1: Synchronized MB-broadcast

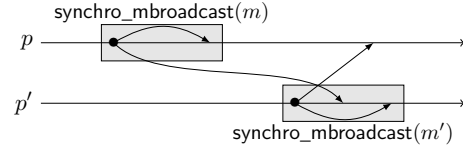


Figure 3: A real-time property

The operation `synchro_mbroadcast()` inherits the properties defining `MBroadcast`, as well as the following property.

**Property 1** *If a process  $p$  returns from the invocation of `synchro_mbroadcast( $m$ )` before  $p'$  invokes `synchro_mbroadcast( $m'$ )`,  $p'$  cannot return from `synchro_mbroadcast( $m'$ )` before it has delivered  $m$ .*

**Proof** Since  $p$  mb-delivers  $m$  before  $m'$ , by the mutual ordering property,  $p'$  must mb-deliver  $m$  before  $m'$ . □ *Property 1*

#### 4.4 Pair Broadcast: MP2 Alone Characterizes `Test&Set()`

**Pair broadcast: Definition** Pair broadcast (in short `PBroadcast`) is a broadcast abstraction that allows a process to broadcast a message that will be delivered at least by all the correct processes. This abstraction provides the processes with two operations denoted `pb_broadcast()` and `pb_deliver()`. When a process invokes `pb_broadcast( $m$ )` we say “it pbroadcasts the message  $m$ ”. The invocation of `pb_deliver()` returns a message  $m$  and we say that a process “pdelivers  $m$ ” or that “ $m$  is pdelivered”. To simplify the presentation (and without loss of generality), it is assumed that all the messages that are pbroadcast are different. `PBroadcast` is defined as the same Validity, No-duplication, Local termination and Global CS-termination properties as `MBroadcast`, and the Pair ordering property (that is a strengthening of the Mutual ordering property):

**Pair ordering.** For any pair of processes  $p$  and  $p'$ , if  $p$  pbroadcasts a message  $m$  and  $p'$  pbroadcasts a message  $m'$ , and if  $m$  and  $m'$  are both pdelivered by  $p$  and  $p'$ , then  $p$  and  $p'$  pdeliver  $m$  and  $m'$  in the same order.

Let us notice that, at the user level, the Pair ordering property only allows the pattern MP2 to occur. So, it prevents both the patterns MP1 and MP3 from occurring. It follows that `PBroadcast` is strictly stronger than Mutual ordering.

**On the computability side of `PBroadcast`.** When only two processes participate in an execution, the pair ordering property implies that all messages are pdelivered by the two processes in the same order. In other words, `PBroadcast` boils down to total-order broadcast in a system composed of only two processes. In particular, it is possible to solve consensus between any pair of processes when `PBroadcast` is available, and consequently the `Test&Set()` operation whose consensus number is 2 [25] (and more generally the objects of the class `Common2` defined in [3]) can be built on top of `PBroadcast`. Appendix C details such a construction. Conversely, Appendix C also presents an algorithm that builds

```

operation mb_broadcast( $m$ ) is                                % code for  $p_i$ 
(1) causal_broadcast INIT( $m$ );
(2) wait (ACK( $m$ ) received from at least  $n - t - 1$  processes);
(3) mb_delivery of  $m$  from  $p_i$ ;
end operation.

when INIT( $m$ ) is causally-delivered from  $p_j$  do
(4) if ( $i \neq j$ ) then
(5)   causal_send ACK( $m$ ) to  $p_j$ ;
(6)   mb_delivery of  $m$  from  $p_j$ 
(7) end if.

```

Algorithm 2: MBroadcast on top of  $\mathcal{CAMP}_{n,t}[t < n/2]$

PBroadcast in the model  $\mathcal{CAMP}_{n,t}[\text{consensus}_2]$  ( $\mathcal{CAMP}_{n,t}[\emptyset]$  enriched with consensus objects available between any pair of processes). Therefore, it is easy to see that PBroadcast has consensus number 2, and is part of the equivalence class Common2.

Unsurprisingly, PBroadcast, that only allows the message pattern MP2 to occur, is computationnally equivalent to the Test&Set() operation, that only allows the memory pattern TS2 to occur. This is similar to the fact that preventing the message pattern MP1 from occurring makes MBroadcast equivalent to shared memory that prevents the memory pattern RW1.

**Remark on trivial extension attempts.** It seems intuitive that, if imposing an order on the messages sent by pairs of processes gives a broadcast with consensus number two, imposing an order on the messages sent by triplets of processes should give a broadcast with consensus number three. Unfortunately, albeit at first glance it seems counter-intuitive, this is not the case. Indeed, let us consider  $n \geq 4$  processes  $p_1, \dots, p_n$  broadcasting respectively messages  $m_1, \dots, m_n$  using an abstraction providing the following guarantee: each triplet of processes  $(p_i, p_j, p_k)$  receives the same first message among  $m_i, m_j$  and  $m_k$ . In particular, it is impossible that all processes receive their own message first, so there is a process  $p_i$  that receives  $m_j$  sent by  $p_j \neq p_i$  as its first message. Remark that  $p_j$  must receive its own message first since receiving  $m_k \neq m_j$  first would violate the property for the triplet  $(p_i, p_j, p_k)$ . Therefore, any process  $p_k$  must receive  $p_j$ 's message first since receiving  $m_\ell \neq m_j$  first would violate the property for the triplet  $(p_j, p_k, p_\ell)$ . This fact can be exploited to solve consensus between  $n$  processes. It follows that imposing an order on the messages sent by triplets of processes provide us with a broadcast whose consensus number is  $\infty$ .

#### 4.5 An implementation of MBroadcast in $\mathcal{CAMP}_{n,t}[t < n/2]$

**Description of the algorithm.** Algorithm 2 implements the MBroadcast abstraction in the system model  $\mathcal{CAMP}_{n,t}[t < n/2]$  ( $\mathcal{CAMP}_{n,t}[\emptyset]$  enriched with the constraint  $t < n/2$ ). When  $p_i$  invokes mb\_broadcast( $m$ ) it issues a causal broadcast of the implementation message INIT( $m$ ), and waits until it has received corresponding acknowledgment from  $(n - t - 1)$  processes. When this occurs it locally mdelivers  $m$ .

When a process  $p_i$  causally delivers a message INIT( $m$ ) from another process  $p_j$ , it causally acknowledges its receipt and locally mdelivers  $m$ .

**Remark on the use of causal ordering.** The causal ordering property is only used to force an order between the ACK messages and the INIT messages. Therefore, practical implementations of MBroadcast do not need to causally order INIT messages with each other. However, the causal ordering of INIT messages is preserved by the algorithm, so Algorithm 2 actually implements causal-MBroadcast. It is also

important to notice that, if the causal\_broadcast of line 1 ensures the Global-CF-Termination property (which characterizes reliable-broadcast, then Algorithm 2 also implements reliable-MBroadcast.

**Remark on the use of quorums.** The necessary and sufficient condition  $t < n/2$  is an operational condition stating there is no partitioning. A more abstract necessary and sufficient condition, has been given in [14], namely the failure detector  $\Sigma$  (see Appendix A) which is the weakest failure detector that allows to build a RW register despite asynchrony and any number of process crashes. Algorithm 2 can be easily modified to work with the quorum-based failure detector  $\Sigma$  instead of the operational assumption  $t < n/2$ .

**Theorem 1** *Algorithm 2 builds MBroadcast in  $\mathcal{CAMP}_{n,t}[t < n/2]$ .*

**Proof** The proof assumes the classical safety and termination properties of causal broadcast. Let  $m_i$  be a message mbroadcast by a process  $p_i$  (line 1). If  $p_i$  is correct, it follows from the termination property of causal broadcast that the message  $\text{INIT}(m_i)$  is received by all correct processes, which are at least  $(n - t - 1)$ . Hence each correct process causally sends back the message  $\text{ACK}(m_i)$  (line 5) and locally mdelivers  $m_i$  (line 6). It then follows from the termination property of causal broadcast that  $p_i$  receives at least  $(n - t - 1)$  messages  $\text{ACK}(m_i)$  and locally mdelivers  $m_i$ . It then follows that the Validity, No-duplication, Local and Global CS-Termination are satisfied.

As far as the mutual ordering property is concerned we have the following. Let  $m_i$  and  $m_j$  be two messages mbroadcast by  $p_i$  and  $p_j \neq p_i$ , respectively. Let us suppose by contradiction that  $p_i$  mdelivers  $m_i$  before  $m_j$ , while  $p_j$  mdelivers  $m_j$  before  $m_i$ . It follows from Lines 4-7 executed by  $p_i$  that  $p_i$  mdelivered  $m_i$  before it causally delivered  $\text{INIT}(m_j)$ . Due to the causal ordering property on the messages  $\text{ACK}(m_i)$  and  $\text{INIT}(m_j)$ , there are at least  $(n - t - 1)$  processes (different from  $p_i$ ) from which  $p_i$  received  $\text{ACK}(m_i)$  that sent  $\text{ACK}(m_i)$  before receiving  $\text{INIT}(m_j)$ . Adding  $p_i$  to these  $(n - t - 1)$  processes, it follows that at least  $(n - t)$  processes received  $\text{INIT}(m_i)$  before  $\text{INIT}(m_j)$ . With the same reasoning, at least  $(n - t)$  processes received  $\text{INIT}(m_j)$  before  $\text{INIT}(m_i)$  for  $p_j$  to mdeliver  $m_j$  before  $p_i$ . But, as  $(n - t) + (n - t) > (n - \frac{n}{2}) + (n - \frac{n}{2}) = n$ , this is impossible, which concludes the proof of mutual ordering.  $\square_{\text{Theorem 1}}$

## 5 MBroadcast versus RW Registers

This section first shows that MBroadcast can be implemented on top of RW registers, and then shows that RW registers can be implemented on top of MBroadcast. Hence, MBroadcast and RW registers have the same computability power. It is important to notice that the algorithms described below are independent of  $t$ .

### 5.1 From regular RW registers to MBroadcast

This section shows that reliable-MBroadcast can be build on top of RW registers in the presence of asynchrony and process crashes. The algorithm only assumes regular registers, hence it also works on top of atomic registers [33].

#### Shared memory and local variables.

- The  $n$  processes share an array of  $n$  SWMR regular registers denoted  $\text{SENT}[1..n]$  such that, for any  $i$ ,  $\text{SENT}[i]$  can be read by any process and written only by  $p_i$ . It contains the (initially empty) list of messages mbroadcast by  $p_i$ . The first message deposited in  $\text{SENT}[i]$  will be in position 1, the second in position 2, etc.

```

operation mb_broadcast( $m$ ) is                                     % code for  $p_i$ 
(1)  $SENT[i] \leftarrow SENT[i] \oplus m$ ;
(2) catch_up();
(3) mb_delivery of  $m$  from  $p_i$ .
end operation.

periodically do catch_up().

internal uninterruptible routine catch_up() is
(4) for  $j$  from 1 to  $i - 1$  and then from  $i + 1$  to  $n$  do
(5)    $msgj \leftarrow SENT[j]$ ;
(6)   for  $k$  from  $delivered_i[j]$  to  $|msgj| - 1$  do mb_delivery of  $msgj[k + 1]$  from  $p_j$  end for;
(7)    $delivered_i[j] \leftarrow |msgj|$ 
(8) end for.

```

Algorithm 3: MBroadcast on top of  $\mathcal{CARW}_{n,t}[\emptyset]$

- Each process  $p_i$  manages an array of local counters (initialized to 0) denoted  $delivered_i$  such that, for any  $j \neq i$ ,  $delivered_i[j]$  contains the number of messages mdelivered from  $p_j$  ( $delivered_i[i]$  is not used).

**Description of the algorithm.** Algorithm 3 builds reliable-MBroadcast on top of  $n$  regular RW registers. When a process  $p_i$  invokes  $mb\_broadcast(m)$  it adds  $m$  at the end of the shared list  $SENT[i]$  ( $\oplus$  stands for concatenation), invokes the internal uninterruptible routine  $catch\_up()$  and then locally mdelivers  $m$ .

The internal routine  $catch\_up()$  is repeatedly invoked to allow  $p_i$  to mdeliver the messages mbroadcast by the other processes.

**Theorem 2** Algorithm 3 builds MBroadcast on top of regular RW registers.

**Proof** Let us consider an execution of Algorithm 3. We prove each property of reliable-mutual-broadcast.

*Proof of the Validity property.* Suppose  $p_i$  mutual-delivers  $m$  from  $p_j$ . If  $i = j$ , this happens on line 3, so  $m$  was mbroadcast by  $p_i$ . Otherwise, it happens on line 6, so  $m$  was read on line 5 and written on line 1 by  $p_j$ , that mbroadcast  $m$ .

*Proof of the No-duplication property.* Suppose  $p_i$  mdelivers twice the  $k^{\text{th}}$  message mbroadcast by  $p_j$ . By Lines 6-7, at the second mdelivery we have  $delivered_i[j] \geq k$ , which is impossible by line 6.

*Proof of the Local termination property.* All operations of Algorithm 3 terminate because they do not contain while loops or recursive calls, and messages that are mbroadcast by  $p_i$  are delivered by  $p_i$  on line 3.

*Proof of the Global CF-Termination property.* Suppose  $p_i$  mdelivers a message  $m$  from  $p_j$ . Then  $p_j$  inserted it in  $SENT[j]$  and never deleted it. Therefore all correct processes eventually mdeliver  $m$  when later they execute  $catch\_up()$ .

*Proof of the Global CS-Termination property.* It is a direct consequence of Global CF-Termination.

*Proof of the Mutual ordering property.* Suppose that  $p_i$  mbroadcasts  $m_i$  and  $p_j$  mbroadcasts  $m_j$ . At least one of the two scenarios must happen:

- $p_i$  completes its write on  $SENT[i]$  (line 1) before  $p_j$  starts its read on  $SENT[i]$  (line 5). In that case, since  $SENT[i]$  is regular,  $p_j$  reads  $m_i$  from  $SENT[i]$  and mdelivers it during its execution of  $catch\_up()$  (line 2), before mdelivering  $m_j$  on line 3.
- Or  $p_j$  completes its write on  $SENT[j]$  (line 1) before  $p_i$  starts its read on  $SENT[j]$  (line 5), so  $p_i$  mdeliver  $m_j$  before  $m_i$ .

□Theorem 2

```

operation write( $v$ ) is                                % code for  $p_i$ 
(1)  synchro_mbroadcast SYNCH();
(2)  let  $t_i$  such that  $\langle t_i, i \rangle > clock_i$ ;
(3)  synchro_mbroadcast WRITE( $v, \langle t_i, i \rangle$ )
end operation.

operation read() is
(4)  synchro_mbroadcast SYNCH();
(5)   $v \leftarrow val_i$ ;
(6)  synchro_mbroadcast WRITE( $v, clock_i$ );
(7)  return( $v$ )
end operation.

when WRITE( $v, c$ ) is mdelivered from  $p_j$  do
(8)  if  $c > clock_i$  then  $val_i \leftarrow v$ ;  $clock_i \leftarrow c$  end if.

```

Algorithm 4: Atomic RW register on top of  $\mathcal{CAMP}_{n,t}[\text{MBroadcast}]$

## 5.2 From MBroadcast to atomic RW registers

Combined with the previous section, this section shows that MBroadcast and atomic RW registers have the same computational power. The algorithm, closely inspired from the ABD algorithm, builds an MWMM atomic register.

**Local variables.** Each process  $p_i$  manages three local variables.

- $val_i$  contains the current value of the register as known by  $p_i$ .
- $t_i$  is the date of the last write issued by  $p_i$ .
- $clock_i$  is a pair (timestamp)  $\langle date, writer \rangle$  defining the identity of the value in  $val_i$  as a timestamp:  $writer$  is the identity of the writer of  $val_i$  and  $date$  the associated Lamport's logical date. Let us remind that all the logical dates are totally ordered according to the usual lexicographical order.

**Description of the algorithm.** Algorithm 4 implements an MWMM atomic register on top of MBroadcast, i.e., in the model  $\mathcal{CAMP}_{n,t}[\text{MBroadcast}]$ .

When  $p_i$  invokes write(), it first mbroadcasts a pure control message SYNCH() to resynchronize its local state with respect to possible write operations that modify the last value of the register (line 1). Then, it computes the timestamp associated with value it wants to write (line 2), and propagates its write operation using a synchronized MBroadcast (line 3).

The read operation is similar. Lines 4-5 provide  $p_i$  with the value it has to return, while line 6 implements the “reads have to write” strategy needed to ensure atomicity of the read operation [7, 9, 38, 44]. Finally, when  $p_i$  mdelivers a WRITE() message, it updates its local context if the new timestamp is higher than the one it currently has in its local variables.

**Theorem 3** *Algorithm 4 implements an atomic RW register on top of  $\mathcal{CAMP}_{n,t}[\text{MBroadcast}]$ .*

**Proof** Let us first remark that Algorithm 4 contains no loop, so all its operations terminate.

Let us consider an execution admitted by Algorithm 4. For each operation  $o$ , we define the timestamp  $ts(o)$  of  $o$  as follows. If  $o$  is a write by  $p_i$ , then  $ts(o) = \langle t_i, i \rangle$  at the end of line 2. If  $o$  is a read by  $p_i$ , then  $ts(o) = clock_i$  at the beginning of line 6. In other words,  $ts(o)$  is the timestamp of the value that is read or written by  $o$ . We also define the binary relation  $\rightarrow$  between operations as  $o_1 \rightarrow o_2$  if either 1)

$o_1$  was terminated before  $o_2$  was started (denoted by  $o_1 \rightarrow_1 o_2$ ), or 2)  $o_2$  is a write and  $ts(o_1) < ts(o_2)$  (denoted by  $o_1 \rightarrow_2 o_2$ ), or 3)  $o_1$  is a write,  $o_2$  is a read, and  $ts(o_1) \leq ts(o_2)$  (denoted by  $o_1 \rightarrow_3 o_2$ ).

Let us first notice that, if  $o_1 \rightarrow o_2$ , then  $ts(o_1) \leq ts(o_2)$ , and if moreover  $o_2$  is a write, then  $ts(o_1) < ts(o_2)$ . This is true by definition for  $\rightarrow_2$  and  $\rightarrow_3$ . For  $\rightarrow_1$ , the first part is a direct consequence of the blocking-mutual-ordering property between the `WRITE` message sent at the end of  $o_1$  (line 3 or 6) and the `SYNCH` message sent at the beginning of  $o_2$  (line 1 or 4). Moreover, if  $o_2$  is a write, then  $ts(o_2) > ts(o_1)$  by line 2.

Let us prove that  $\rightarrow$  is cycle-free. Indeed, suppose there is a cycle  $o_1 \rightarrow o_2 \rightarrow \dots \rightarrow o_k = o_1$  containing at least two operations. By what precedes, all operations in the cycle have the same timestamp, hence there cannot be any write operation. Moreover, there cannot be only reads, because they would be ordered only by  $\rightarrow_1$  which is cycle-free.

Finally, the reflexive and transitive closure of  $\rightarrow$  can be extended into a total order that respects real time thanks to  $\rightarrow_1$ , and such that any read returns the initial value if its timestamp is  $\langle 0, 0 \rangle$ , or the value written by the preceding write since  $val_i$  and  $clock_i$  are updated jointly on line 8, thanks to  $\rightarrow_2$  and  $\rightarrow_3$ . Hence, the algorithm is linearizable.

□ *Theorem 3*

Suppressing Line 1 (resp. Line 6) builds an SWMR atomic register (resp. an SWMR regular register). Algorithm 4 can also be easily adapted to work with the four types of MWMR regular registers defined in [51].

### 5.3 A remark on complexity

Although Algorithms 3 and 4 prove that mbroadcast and atomic RW registers have the same computability power, the same cannot be said from a complexity point of view, since  $n$  single-writer multi-reader atomic registers must be used and read in Algorithm 3 to implement MBroadcast.

In fact, this complexity is necessary, even with the use of multi-writer multi-reader atomic registers. It was proven in [28], that at least  $n$  multi-writer multi-reader atomic registers are necessary to implement an array of one single-writer multi-reader atomic register per process. This lower bound also applies to MBroadcast since, if  $k < n$  registers were sufficient to implement MBroadcast, Algorithm 3 could, in turn, be used to simulate the array of  $n$  single-writer multi-reader registers. This justifies our introduction of a new abstraction, that allows algorithms with better complexities than read/write registers.

### 5.4 What is actually needed to build a RW register

It follows from the previous results that the operational condition

$$(\text{MP2} \vee \text{MP3}) \text{ or } (t < n/2)$$

is necessary and sufficient to build an atomic RW register on top of a crash-prone asynchronous message-passing system. It is worth noticing that the first sub-condition  $\text{MP2} \vee \text{MP3}$  is on the messages exchanged by each pair of processes (and, for any pair of processes, can change from one message exchange to another one without being explicitly known by the processes) while the other one  $t < n/2$  is on global system parameters.

What is captured by  $\text{MP2} \vee \text{MP3}$  is the fact that, for each pair of processes, as soon as one of them does not ignore the message from the other one (which is not a pattern captured by  $\text{MP1}$ ), it is possible to build an atomic RW register.

**Remark: From consensus on pairs of processes to multi-writer multi-reader registers.** Since PBroadcast is stronger than MBroadcast, the implementation of PBroadcast in the model  $\mathcal{CAMP}_{n,t}[\text{consensus}_2]$  presented in Appendix C also provides an implementation of MBroadcast that can be exploited to implement an atomic register (using Algorithm 4), in any message-passing system where consensus is

```

operation acquire() is                                % code for  $p_i$ 
(1)  fifo_synchro_mbroadcast HELLO();
(2)  let  $t_i$  such that  $\langle t_i, i \rangle > \max(ticket_i)$ ;
(3)  fifo_synchro_mbroadcast TICKET( $t_i$ );
(4)  wait( $\langle t_i, i \rangle = \min(ticket_i)$ )
end operation.

operation release() is
(5)  fifo_broadcast GOODBYE( $t_i$ )
end operation.

when HELLO() is fifo-mdelivered from  $p_j$  do
(6)   $ticket_i \leftarrow ticket_i \cup \langle 0, j \rangle$ .

when TICKET( $t$ ) is fifo-mdelivered from  $p_j$  do
(7)   $ticket_i \leftarrow (ticket_i \setminus \{ \langle 0, j \rangle \}) \cup \{ \langle t, j \rangle \}$ .

when GOODBYE( $t$ ) is fifo-delivered from  $p_j$  do
(8)   $ticket_i \leftarrow ticket_i \setminus \{ \langle t, j \rangle \}$ .

```

Algorithm 5: An MBroadcast-based version of Lamport’s Bakery algorithm

available between any pair of processes. In particular, the assumption that  $t < \frac{n}{2}$  is not required in this case. Remark that this fact could have been previously established by using the consensus between two processes to implement single-reader single-writer registers, that have the same computability power as multi-writer multi-reader registers [33], but, to our knowledge, it had never been stated explicitly.

## 6 MBroadcast in Action: Mutex

**Preliminary remark.** As announced in the Introduction, this section and the next section illustrate uses of the MBroadcast abstraction, where design simplicity is considered as a first class criterion. As already said, it is important to notice that none of the algorithms presented below is based on quorums. Moreover, as the reader will see, the mutex algorithm and the consensus algorithm have the same structure as Algorithm 4.

### 6.1 Mutex

Considering the asynchronous message-passing system (i.e. the computing model  $\mathcal{CAM}\mathcal{P}_{n,t}[\emptyset]$ ) this section addresses the mutual exclusion problem (mutex). This problem was introduced by E.W. Dijkstra in 1965 [17]. From a historical point of view it is the first distributed computing problem (see [49] for a historical survey on RW-based mutex algorithms). It is defined by two operations denoted `acquire()` and `release()`, that are used to bracket a section of code, called *critical section*, such that the following properties are satisfied.

- Safety. At most one process at a time can be executing the critical section.
- Liveness. If a process invokes `acquire()`, it eventually enters the critical section.

One of the most famous mutex algorithms is the Bakery algorithm due to L. Lamport [31, 35]. This algorithm is based on non-atomic RW registers. This section presents an MBroadcast-based re-writing of this algorithm suited to the asynchronous message-passing communication model.

### 6.2 An MBroadcast-based rewriting of Lamport’s Bakery algorithm

**Local variables.** Each process  $p_i$  manages two local variables.



- An initially empty set  $ticket_i$  that will contain timestamps  $\langle t, id \rangle$  where  $t$  is a ticket number and  $id$  a process identity.
- The variable  $t_i$  contains the last ticket number used by  $p_i$ .

**Description of the algorithm.** Algorithm 5 is an MBroadcast-based version of Lamport’s Bakery algorithm [31]. When a process  $p_i$  invokes  $acquire()$ , it first invokes  $fifo\_synchro\_mbroadcast\ HELLO()$  to make public the fact that it starts competing to access the critical section (line 1). When this synchronized  $fifo\_synchro\_mbroadcast()$  invocation terminates,  $p_i$  knows that processes that will start competing afterwards are informed it started competing. Process  $p_i$  then computes a ticket number higher than all the ticket numbers it knows (line 2) and invokes again  $fifo\_synchro\_mbroadcast$  to inform the processes that its request for the critical section is timestamped  $\langle ticket_i, i \rangle$  (line 3). Finally,  $p_i$  waits until its request has the smallest timestamp (according to lexicographical order) (line 4).

When a process  $p_i$   $fifo$ -mdelivers the message  $HELLO()$  from process  $p_j$ , it adds the pair  $\langle 0, j \rangle$  to its set  $ticket_i$ , which registers the requests of all the competing processes as known by  $p_i$  (line 6);

When a process  $p_i$   $fifo$ -mdelivers the message  $TICKET(t)$  from process  $p_j$ , it replaces the pair  $\langle 0, j \rangle$  by the pair  $\langle t, j \rangle$  in its local  $ticket_i$ , which now stores the request of  $p_j$  with its competing timestamp (line 7).

When a process  $p_i$  invokes  $release()$ , it invokes  $synchro\_mbroadcast\ GOODBYE()$  to inform the other processes it is no longer interested in the critical section. (line 5). Consequently when a process  $p_i$   $fifo$ -delivers a message  $GOODBYE()$  from a process  $p_j$ , it updates accordingly its local set of requests  $ticket_i$  (line 8).

**Theorem 4** *Considering the system model  $CAMP_{n,t}[MBroadcast]$ , Algorithm 5 ensures that there is at most one process at a time in the critical section (safety), and that if no process crashes while executing  $acquire()$ ,  $release()$ , or the code inside the critical section, then all invocations of  $acquire()$  and  $release()$  terminate (liveness).*

**Proof** *Proof of the safety property.* Suppose, by contradiction, that two processes  $p_i$  and  $p_j$  are in the critical section at the same time, and let  $t_i$  and  $t_j$  be their respective order values after line 2. Without loss of generality, let us suppose that  $\langle t_i, i \rangle < \langle t_j, j \rangle$ . Two cases are consistent with the Mutual Ordering property applied to  $p_i$ ’s  $HELLO()$  message and  $p_j$ ’s  $TICKET(t_j)$  message.

- If  $p_i$  receives  $TICKET(t_j)$  before it  $fifo$ -mbroadcasts  $HELLO()$ , then  $\langle t_j, j \rangle \in ticket_i$  after line 2 thanks to line 7, and, due to  $fifo$  ordering, this will remain true as long as  $p_j$  remains in critical section. Then,  $p_i$  picks  $t_i$  such that  $\langle t_i, i \rangle > \langle t_j, j \rangle$  (line 2). This is in contradiction with the fact that  $\langle t_i, i \rangle < \langle t_j, j \rangle$ .
- If  $p_j$   $fifo$ -mdelivers  $HELLO()$  from  $p_i$  before it  $fifo$ -mbroadcasts  $TICKET(t_j)$ , then, when  $p_j$  entered the critical section, either  $\langle 0, i \rangle \in ticket_j$  (by line 6) or  $\langle t_i, i \rangle \in ticket_j$  (by line 7), due to the  $fifo$  ordering. In both cases, this contradicts the fact that  $\langle t_j, j \rangle = \min(ticket_j)$  (line 4).

*Proof of the liveness property.* Let us observe that the  $release()$  operation has no loop, and the  $acquire()$  operation has a single  $wait()$  statement (line 4). Suppose, by contradiction, that some process  $p_i$  remains forever blocked at line 4. Without loss of generality, let us assume that its timestamp is the smallest one, (i.e. all processes with a lower timestamp eventually entered and exited the critical section).

After all correct processes have received the message  $TICKET(t_i)$  sent by  $p_i$  (line 3), all processes  $p_j$  that execute line 2 pick a value  $t_j$  such that  $\langle t_j, j \rangle > \langle t_i, i \rangle$ . In other words, a finite number of timestamps  $\langle t_j, j \rangle < \langle t_i, i \rangle$  are ever picked in the execution, and, by minimality of  $\langle t_i, i \rangle$ , all these timestamps let their process enter the critical section. As all critical sections terminate, they also all leave critical

section and send a GOODBYE() message, that is eventually received by  $p_i$ . After  $p_i$  has received all these messages, a finite number of processes  $p_j$  will call acquire() again and pick  $t_j$  such that  $\langle t_j, j \rangle > \langle t_i, i \rangle$ . When  $p_i$  has received all the respective  $ticket(t_j)$  messages, it will have  $\langle t, i \rangle = \min(ticket_i)$ . A contradiction.  $\square$ Theorem 4

**Remark.** Using the failure detector introduced in [16], it is possible to implement mutex in the presence of process crashes occurring while a process is executing acquire(), release(), or the code inside the critical section.

## 7 MBroadcast in Action: Consensus

The consensus problem was introduced by Lamport, Shostak and Pease [36, 42] in the context of synchronous systems prone to Byzantine process failures. As stated previously, here we consider it in the context of asynchrony and process crashes. Let us recall that consensus is a fundamental problem that lies at the center of the set of distributed agreement problems.

### 7.1 Definition

Consensus is defined by a single one-shot operation denoted propose() that takes a value as input parameter and returns a value as result. When a process invokes propose( $v$ ), we say it proposes  $v$ . If the returned value is  $w$ , we say the process decides  $w$ . Consensus is defined by the following three properties. Validity: If a process decides  $v$ , some process proposed  $v$ . Agreement: No two processes decide different values. Termination: If a process does not crash, it decides a value.

### 7.2 Enriching the model with additional computability power

It is well-known that consensus cannot be solved in asynchronous distributed systems where even a single process may crash, be the underlying communication medium message-passing [20] or RW registers [37]. We consider here that this additional computability power is given by the failure detector denoted  $\Omega$ , which is the weakest failure detector with which consensus can be solved [11].

$\Omega$  provides the processes with a single operation denoted leader(). This operation has no input, and each of its invocations returns a process identifier. It is defined by the following property.

- Eventual leadership. In any execution, there exists a process identifier  $j$  such that (1)  $p_j$  is a correct process and (2) the number of times leader() returns an identifier  $k \neq j$  to any process is finite.

### 7.3 An MBroadcast-based variant of the Paxos consensus algorithm

$\perp$  denotes a default value that cannot be proposed to consensus.

**Local variables.** Each process  $p_i$  manages three local variables.

- $decided_i$ , initialized to  $\perp$ , will contain the decided value.
- $t_i$  is a scalar logical time (its initial value is irrelevant). Each round of the algorithm initiated by  $p_i$  is uniquely identified by a timestamp  $\langle t_i, i \rangle$ , that plays the same role as the ballot number in Lamport's article. Recall that any two such pairs can be ordered by lexicographical order.
- $clock_i$ , initialized to  $\langle 0, 0 \rangle$ , contains the timestamp identifying the current round.

```

operation propose( $v_i$ ) is                                     % code for  $p_i$ 
(1)  $val_i \leftarrow v_i$ ;
(2) while ( $decided_i = \perp$ ) do
(3)   if ( $leader() = i$ ) then
(4)     let  $t_i$  such that  $\langle t_i, i \rangle > clock_i$ ;
(5)     synchro_mbroadcast BEGIN( $t_i$ );
(6)     if  $\langle t_i, i \rangle = clock_i$  then synchro_mbroadcast VOTED( $t, val_i$ ) end if;
(7)     if  $\langle t_i, i \rangle = clock_i$  then reliable_broadcast SUCCESS( $val_i$ ) end if
(8)   end if
(9) end while;
(10) return( $decided_i$ )
end operation.

when BEGIN( $t$ ) is mdelivered from  $p_j$  do
(11) if  $\langle t, j \rangle > clock_i$  then  $clock_i \leftarrow \langle t, j \rangle$  end if.

when VOTED( $t, v$ ) is mdelivered from  $p_j$  do
(12) if  $\langle t, j \rangle > clock_i$  then  $clock_i \leftarrow \langle t, j \rangle$ ;  $val_i \leftarrow v$  end if.

when SUCCESS( $v$ ) is reliable-delivered from  $p_j$  do
(13)  $decided_i \leftarrow v$ .

```

Algorithm 6: An MBroadcast-based variant of Lamport's Paxos algorithm

**Description of the algorithm.** Algorithm 6 solves the consensus problem in the system model  $\mathcal{CAMP}_{n,t}[\text{MBroadcast}, \Omega]$ . When a process  $p_i$  invokes  $\text{propose}(v_i)$  it first initializes  $val_i$  (its current local estimate of the decided value) to  $v_i$  (line 1). It then enters a loop it will exit when  $decided_i$  is different from  $\perp$  (lines 2-9). If  $decided_i \neq \perp$ , it decides it (line 10). Otherwise,  $p_i$  checks if it is the leader by calling  $leader()$  on  $\Omega$  (line 3). If it is not, it re-enters the while loop. If  $leader() = i$ ,  $p_i$  competes to impose its current estimate to be the decided value. To this end, it first computes a new timestamp  $\langle t_i, i \rangle$  greater than any timestamp it knows (line 4) (this timestamp identifies its current competition to impose a decided value) and invokes  $\text{synchro\_mbroadcast}$  BEGIN( $t_i$ ) to inform the other processes it starts competing (line 5). Process  $p_i$  then checks if the timestamp  $\langle t_i, i \rangle$  is equal to  $clock_i$  (line 6). If it is not the case, it aborts the competition. Otherwise, it invokes  $\text{synchro\_mbroadcast}$  VOTED( $t, val_i$ ) (line 7) to inform the processes that it champions  $val_i$  timestamped  $\langle t_i, i \rangle$ . Then, it checks again that  $\langle t_i, i \rangle = clock_i$ , and aborts its competition if it is not. It then discovers that it has won the competition and informs the other processes that  $val_i$  is the decided value (line 7). Let us notice that between its reads at line 6 and line 7, the value of  $clock_i$  may have been modified.

When  $p_i$  mdelivers BEGIN( $t$ ) or VOTED( $t, v$ ) from a process  $p_j$ , it updates its local variables accordingly. It does the same when it delivers the message SUCCESS( $v$ ).

**Theorem 5** *Algorithm 6 solves consensus in the system model  $\mathcal{CAMP}_{n,t}[\text{MBroadcast}, \Omega]$ .*

**Proof** *Proof of the Validity property.* Only proposed values or  $\perp$  are ever contained in  $val_i$  and  $decided_i$ , or carried by VOTED( $-, v$ ) and SUCCESS( $v$ ), and the decided value cannot be  $\perp$  by line 2.

*Proof of the Agreement property.* Due to lines 10 and 13, a process can only decide a value that was carried by a SUCCESS( $v$ ) message. Let us consider the process  $p_i$  that broadcast a message SUCCESS( $v_i$ ) when its variable  $t_i = t$ , such that  $t_i$  is minimal. Let us prove that, for all messages VOTED( $t_j, v_j$ ) such that  $t_j > t_i, v_j = v_i$ . Let us suppose, by contradiction, that this is not the case, and let us consider such a message with a minimal  $t_j$ . Due to lines 7 and 11,  $p_i$  had not yet received BEGIN( $b_j$ ) mbroadcast by  $p_j$ , before mdelivering its own message VOTED( $t_i, v_i$ ). So, by the mutual ordering property,  $p_j$  mdelivered VOTED( $t_i, v_i$ ) before BEGIN( $t_j$ ). By minimality of  $t_j$ , this implies that  $val_j = v_i$  when  $p_j$  executes line 6, which contradicts the fact that  $v_j \neq v_i$ .

Let us consider a message SUCCESS( $v_k$ ) broadcast by a process  $p_k$ , and let  $t_k$  its value of  $t$  at that time. By the minimality of  $t_i$ , we have  $t_k \geq t_i$ . By what precedes,  $p_k$  mbroadcast VOTED( $t_k, v_i$ ) at

line 6, and did not deliver any message  $\text{VOTED}(t_l, v_l)$  with  $t_l > t_i$  and  $v_l \neq v_i$ . As  $val_k$  is only updated at line 12,  $val_k = v_i$  at line 7 and  $v_k = v_i$ . Therefore, no two processes decide different values.

*Proof of the Termination property.* By the definition of  $\Omega$ , there is a time after which only one process  $p_i$ , that is correct, will execute the lines 4-7. In particular, after some finite time,  $p_i$  will choose a round (ballot) number  $\langle t_i, i \rangle$ , greater than all other round (ballot) numbers chosen by other processes. After line 5,  $\langle t_i, i \rangle = \text{clock}_i$ , so  $p_i$  broadcasts a message  $\text{SUCCESS}(v)$  at line 7. At the delivery of this message, all the non-crashed processes  $p_j$  will set their variable  $\text{decided}_j$  to some non- $\perp$  value and terminate their execution.  $\square$ Theorem 5

## 8 Conclusion

Having a better understanding of the power and weaknesses of the basic communication mechanisms provided by asynchronous crash-prone distributed systems is a central issue of distributed computing. The aim of this article was to be a step in such an approach. To this end, the article investigated basic relations linking send/receive message patterns and read/write patterns. It introduced three basic message patterns, each involving a pair of processes, and showed that only two of these patterns have a RW counterpart. This gives a new and deeper view of the different ways processes communicate (and consequently cooperate) in RW systems and in message-passing systems. Then the article introduced a new message-passing communication abstraction denoted Mutual Broadcast, the computability power of which is the same as the one of RW registers. Its main property (called *mutual ordering*) is the fact that, for each pair of processes  $p$  and  $p'$ , if  $p$  broadcasts a message  $m$  and  $p'$  broadcasts a message  $m'$ , it is not possible for  $p$  to deliver first (its message)  $m$  and then  $m'$  while  $p'$  delivers first (its message)  $m'$  and then  $m$ . In a very interesting way, it appears that this implicit synchronization embedded in the MBroadcast abstraction allows for the design of simple algorithms in which, among other properties, no notion of quorums is explicitly required. The simplicity of MBroadcast-based algorithms has been highlighted with examples including simple re-writing of existing algorithms such as Lamport's Bakery and Paxos algorithms.

Nevertheless the quest for the Grail of a deeper understanding of message-passing systems is not complete. On the side of shared memory systems an apex has been attained with Herlihy's consensus hierarchy and its extensions [2, 13, 25, 43, 46]. The same has not yet been achieved for message-passing systems: is there a consensus hierarchy based on specific broadcast abstractions? If the answer is "yes", which is this hierarchy? The case for the consensus number  $\infty$  is total order broadcast [12], the case for the consensus number 2 is PBroadcast, but no specific broadcast abstraction is known for each consensus number  $x \in [3..+\infty)$ . We conjecture that there are no such specific broadcast abstractions.<sup>3</sup>

## References

- [1] Afek Y., Attiya H., Dolev D., Gafni E., Merritt M. and Shavit N., Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873-890 (1993)
- [2] Afek Y., Ellen F., and Gafni E., Deterministic objects: life beyond consensus. *Proc. 35th ACM Symposium on Principles of Distributed Computing (PODC'16)*, ACM Press, pp. 97-106 (2016)
- [3] Afek Y., Weisberger E., and Weisman H., A completeness theorem for a class of synchronization objects. *Proc. 12th ACM Symposium on Principles of Distributed Computing (PODC'93)*, ACM Press, pp. 159-170 (1993)

---

<sup>3</sup>Let us remind that, when considering failure detectors, it has been shown in [14] that the weakest failure detector to implement Test&Set and Compare&Swap in asynchronous message-passing systems prone to any number of crashes is the same failure detector (namely,  $\Omega$ ).

- [4] Ahamad M., Neiger G., Burns J.E., Hutto P.W., and Kohli P., Causal memory: definitions, implementation and programming. *Distributed Computing*, 9:37-49 (1995)
- [5] Anderson J., Multi-writer composite registers. *Distributed Computing*, 7(4):175-195 (1994)
- [6] Attiya H., Efficient and robust sharing of memory in message-passing systems. *Journal of Algorithms*, 34:109-127 (2000)
- [7] Attiya H., Bar-Noy A. and Dolev D., Sharing memory robustly in message passing systems. *Journal of the ACM*, 42(1):121-132 (1995)
- [8] Attiya H., Herlihy M., and Rachman O., Atomic snapshots using lattice agreement. *Distributed Computing*, 8:121-132 (1995)
- [9] Attiya H. and Welch J., *Distributed computing: fundamentals, simulations and advanced topics*, (2d Edition), Wiley-Interscience, 414 pages (2004)
- [10] Birman K.P. and Joseph T.A., Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47-76 (1987)
- [11] Chandra T.D., Hadzilacos V., and Toueg S., The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685-722 (1996)
- [12] Chandra T.D. and Toueg S., Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225-267 (1996)
- [13] Daian E., Losa G., Afek Y., and Gafni E., A wealth of sub-consensus deterministic objects. *Proc. 32nd International Symposium on Distributed Computing (DISC'18)*, LIPICs 121, Article 17, 17 pages (2018)
- [14] Delporte-Gallet C., Fauconnier H., and Guerraoui R., Tight failure detection bounds on atomic object implementations. *Journal of the ACM*, 57(4), Article 22, 32 pages (2010)
- [15] Delporte C., Fauconnier H., Rajsbaum S., and Raynal M., Implementing snapshot objects on top of crash-prone asynchronous message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 29(9):2033-2045 (2018)
- [16] Delporte C., Fauconnier H. and Raynal M., On the weakest information on failures to solve mutual exclusion and consensus in asynchronous crash prone read/write systems. *Journal of Parallel and Distributed Computing*, 153:110-118 (2021)
- [17] Dijkstra E.W., Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569 (1965)
- [18] Dutta P., Guerraoui R., Levy R., and Vukolic M., Fast access to distributed atomic memory. *SIAM Journal of Computing*, 39(8):3752-3783 (2010)
- [19] Faleiro J.M., Rajamani S., Rajan K., Ramalingam G., and Vaswani K., Generalized lattice agreement. *Proc. 31th ACM Symposium on Principles of Distributed Computing (PODC'12)*, ACM Press, pp. 125-134 (2012)
- [20] Fischer M.J., Lynch N.A. and Paterson M.S., Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374-382 (1985)
- [21] Fischer M.J. and Merritt M., Appraising two decades of distributed computing theory research. *Distributed Computing*, 16(2-3):239-247 (2003)
- [22] Freitas de Souza L., Kuznetsov P., Rieutord Th., and Tucci Piergiovanni S., Accountability and Reconfiguration: Self-Healing Lattice Agreement. *Proc. 25th Int'l Conference on Principles of Distributed Systems (OPODIS 2021)*, LIPICs Vol. 217, pp. 25:1-25:23 (2021)
- [23] Georgiou Ch., Hadjistasi T., Nicolaou N., and Schwarzmann A., Implementing three exchange read operations for distributed atomic storage. *Journal of Parallel Distributed Computing*, 163: 97-113 (2022)

- [24] Hadzilacos V. and Toueg S., A modular approach to fault-tolerant broadcasts and related problems. *Tech Report 94-1425*, 83 pages, Cornell University (1994), Extended version of “Fault-Tolerant Broadcasts and Related Problems”, in *Distributed systems, 2nd Edition*, Addison-Wesley/ACM, pp. 97-145 (1993)
- [25] Herlihy M. P., Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124-149 (1991)
- [26] Herlihy M., Rajsbaum S., Raynal M., and Stainer J., From wait-free to arbitrary concurrent solo executions in colorless distributed computing. *Theoretical Computer Science*, 683:1-21 (2017)
- [27] Herlihy M.P. and Wing J.M., Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463-492 (1990)
- [28] Imbs D., Kuznetsov P., and Rieutord Th., Progress-space tradeoffs in single-writer memory implementations. *21st Int’l Conference on Principles of Distributed Systems (OPODIS)*, LIPIcs Vol.295, pp. 9:1-9:17 (2017)
- [29] Imbs D., Mostéfaoui A., Perrin M., and Raynal M., Which broadcast abstraction captures  $k$ -set agreement? *Proc. 31th Int’l Symposium on Distributed Computing (DISC’17)*, LIPIcs Vol. 91, Article 27, 16 pages (2017)
- [30] Imbs D., Mostéfaoui A., Perrin M., and Raynal M., Set-constrained delivery broadcast: a communication abstraction for read/write implementable distributed objects. *Theoretical Computer Science*, 886:49-68 (2021)
- [31] Lamport L., A new solution of Dijkstra’s concurrent programming problem. *Communications of the ACM*, 17(8):453-455 (1974)
- [32] Lamport L., Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558-565 (1978)
- [33] Lamport L., On interprocess communication, Part I: basic formalism. *Distributed Computing*, 1(2):77-85 (1986)
- [34] Lamport L., The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133-169 (1998)
- [35] Lamport L., Deconstructing the Bakery to build a distributed state machine. *Communications of the ACM*, 65(9):58-66 (2022)
- [36] Lamport L., Shostack R., and Pease M., The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382-401 (1982)
- [37] Loui M.C., and Abu-Amara H.H., Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163-183, JAI Press (1987)
- [38] Lynch N.A., *Distributed algorithms*. Morgan Kaufmann, 872 pages (1996)
- [39] Misra A. and Kshemkalyani A., Solvability of Byzantine fault-tolerant causal ordering problems. *Proc. 10th Int’l Conference on Networked Systems (NETYS 22)*, Springer LNCS 13464, pp. 87-103 (2022)
- [40] Mostéfaoui A. and Raynal M., Two-bit messages are sufficient to implement atomic read/write registers in crash-prone systems. *Proc. 35th ACM Symposium on Principles of Distributed Computing (PODC’16)*, ACM Press, pp. 381-390 (2016)
- [41] Mostéfaoui, Raynal M., and Roy M., Time-efficient read/write register in crash-prone asynchronous message-passing systems. *Computing*, 10(1):3-17 (2019)
- [42] Pease M., Shostak R., and Lamport L., Reaching agreement in the presence of faults. *Journal of the ACM*, 27:228-234 (1980)

- [43] Perrin M., Mostéfaoui A., Bonin G., Courtillat-Piazza L., Extending the wait-free hierarchy to multi-threaded systems. *Distributed Computing*, 35(4):375-398 (2022)
- [44] Raynal M., *Concurrent programming: algorithms, principles and foundations*. Springer, 515 pages (2013)
- [45] Raynal M., *Fault-tolerant message-passing distributed systems: an algorithmic approach*. Springer, 459 pages (2018)
- [46] Raynal M., *Concurrent crash-prone shared memory systems: a few theoretical notions*. Morgan & Claypool Publishers, 115 pages (2022)
- [47] Raynal M., Mutual exclusion vs consensus: both sides of the same coin? *Bulletin of the European Association of Theoretical Computer Science (EATCS)*, Vol.140, 14 pages (2023)
- [48] Raynal M., Schiper A., and Toueg S., The causal ordering abstraction and a simple way to simple to implement it. *Information Processing Letters*, 39(6):343-350 (1991)
- [49] Raynal M. and Taubenfeld G., A visit to mutual exclusion in seven dates. *Theoretical Computer Science*, 919:47-65 (2022)
- [50] Ruppert E., Implementing shared registers in asynchronous message-passing systems. *Springer Encyclopedia of Algorithms*, pp. 400-403 (2008)
- [51] Shao Ch., Welch J.L., Pierce E., and Lee H., Multiwriters consistency conditions for shared memory registers. *SIAM Journal of Computing*, 40(1):28-62 (2011)

## A The failure Detector $\Sigma$

The failure detector denoted  $\Sigma$ , introduced in [14], is the weakest failure detector that allows to build an atomic RW register.  $\Sigma$  is a quorum-based failure detector (which can be trivially implemented in  $\mathcal{CAMP}_{n,t}[t < n/2]$ ). It provides each process with an operation denoted `quorum()` which returns a non-empty set of processes with the following properties.

- Local Termination (Liveness): Any invocation of `quorum()` by a correct process terminates.
- Intersection (Safety): If an invocation of `quorum()` returns *set1* and another invocation of `quorum()` returns *set2*, we have  $set1 \cap set2 \neq \emptyset$ .
- Global Termination (Convergence): There a finite time after which any invocation of `quorum()` contains only correct processes.

## B MBroadcast in Action: Lattice Agreement

One of the very first articles on the use of lattices to solve distributed computing problems is [8] where a snapshot object is built from a lattice data structure. Later developments appeared in [19]. More recently, lattice agreement has been used as a building block to solve accountability and reconfiguration issues encountered in distributed computing [22].

### B.1 Definition

A bounded join-semilattice  $(L, \perp, \sqsubseteq, \sqcup)$  is composed of a set  $L$  of elements partially ordered according to a relation  $\sqsubseteq$ , such that there is a smallest element  $\perp$  and for all  $x, y \in L$ , there exists a least upper bound of  $x$  and  $y$ , denoted by  $x \sqcup y$ .

Lattice agreement is similar to consensus, namely each process may propose a value and decide a value. It is defined by the following properties.

```

operation propose( $v_i$ ) is                                % code for  $p_i$ 
(1) repeat  $prev_i \leftarrow view_i$ ;
(2)     synchro_mbroadcast STATE( $view_i \sqcup v_i$ )
(3) until ( $prev_i = view_i$ );
(4) return( $view_i$ )
end operation.

when STATE( $v$ ) is mdelivered from  $p_j$  do
(5)  $view_i \leftarrow view_i \sqcup v$ .

```

Algorithm 7: Lattice agreement in  $\mathcal{CAMP}_{n,t}[\text{MBroadcast}]$

**Validity.** The value decided by a process  $p_i$  is the least upper bound of a subset of the proposed values and contains the value proposed by  $p_i$ .

**Consistency.** If  $p_i$  decides  $x_i$  and  $p_j$  decides  $x_j$ , then  $x_i \sqsubseteq x_j$  or  $x_j \sqsubseteq x_i$ .

**Termination.** If a process does not crash, it decides a value.

## B.2 An MBroadcast-based lattice agreement algorithm

**Local variables.** Each process  $p_i$  manages two local variables.

- $view_i$  contains the current view of the value that  $p_i$  will return. It is initialized to  $\perp$ .
- $prev_i$  is an auxiliary variable. g

**Description of the algorithm.** Algorithm 7 solves lattice agreement in the system model  $\mathcal{CAMP}_{n,t}[\text{MBroadcast}]$ . It is based on the classical double-scan principle. When a process  $p_i$  invokes  $\text{propose}(v_i)$  it repeatedly mbroadcasts its current view of the decided value enriched with the value  $v_i$  it proposes, until  $view_i$  has not been modified since the previous mbroadcast. When it mdelivers a message STATE( $v$ ),  $p_i$  updates its local view  $view_i$ . Let us notice that  $view_i$  can be updated before  $p_i$  invokes  $\text{propose}(v_i)$ .

## B.3 Proof of the lattice agreement algorithm

**Theorem 6** Algorithm 7 solves lattice agreement in the system model  $\mathcal{CAMP}_{n,t}[\text{MBroadcast}]$ .

**Proof** *Proof of the Validity property.* Let us notice that  $view_i$  is updated only at line 5 as the upper bound of values  $p_i$  has mdelivered, and those values have been built at line 2 by their sender. It follows that, at any time, a local variable  $view_i$  contains only an upper bound of proposed values. Moreover, it follows from the use of the synchronized MBroadcast that  $p_i$  can terminate its invocation of  $\text{propose}(v_i)$  only after it receives a view including  $v_i$ . The validity property follows.

*Proof of the Consistency property.* Let us suppose  $p_i$  proposes  $v_i$  and decides  $x_i$ , and  $p_j$  proposes  $v_j$  and decides  $x_j$ . By the Validity property,  $v_i \sqsubseteq x_i$  and  $v_j \sqsubseteq x_j$ . So, just before deciding,  $p_i$  and  $p_j$  mbroadcast  $x_i = x_i \sqcup v_i$  and  $x_j = x_j \sqcup v_j$ . By the mutual ordering property,  $p_i$  mdelivered the message from  $p_j$  before its own, in which case, by lines 2 and 5,  $x_i = view_i = view_i \sqcup x_j = x_i \sqcup x_j$ , i.e.  $x_j \leq x_i$ , or  $p_j$  mdelivered the message from  $p_i$  before its own, in which case  $x_i \leq x_j$ .

*Proof of the Termination property.* Due to line 5,  $view_i$  can only grow and by line 3, the value of  $view_i$  between two iterations of the loop must be different. As seen in the proof of the Validity property,  $view_i$  can only take values obtained as a least upper bound of proposed values, which are in



```

operation  $CONSENSUS\{i, j\}.propose(v)$  is           % code for  $p_i$ 
(1)  synchro_pbroadcast PROPOSITION( $v, \{i, j\}$ );
(2)  return( $decided\{i, j\}_i$ )
end operation.

when PROPOSITION( $v, \{i, j\}$ ) is pdelivered from  $p_i$  or  $p_j$  do
(3)  if  $decided\{i, j\}_i = \perp$  then  $decided\{i, j\}_i \leftarrow v$  end if.

```

Algorithm 8: Consensus between two processes  $p_i$  and  $p_j$  in  $\mathcal{CAMP}_{n,t}[\text{PBroadcast}]$

finite number. Therefore, if  $N$  values are proposed in an execution,  $view_i$  can take at most  $N$  different values at line 3. It follows that the number of iterations is finite.  $\square_{\text{Theorem 6}}$

## C On the Computability Side: MP2 Alone Characterizes Test&Set()

### C.1 From PBroadcast to two-process consensus

Based on PBroadcast, Algorithm 7 implements consensus between any pair of processes  $p_i$  and  $p_j$ . Each process  $p_i$  manages a local variable  $decided\{i, j\}_i$  that is initialized to a default value  $\perp$  that cannot be proposed. The set-based notation is used to distinguish the different two-process consensus in which  $p_i$  can be involved e.g.,  $decided\{i, j\}_i$  and  $decided\{i, k\}_i$ .

**Description and proof of the algorithm** Given a pair of distinct processes  $(p_i, p_j)$ , let  $CONSENSUS\{i, j\}$  denote the consensus object that is shared by the set of processes  $\{i, j\}$ . So, at the upper level, this object is accessed only by  $p_i$  and  $p_j$  by invoking  $CONSENSUS\{i, j\}.propose()$ . When  $p_i$  invokes  $CONSENSUS\{i, j\}.propose(v)$ , it executes a synchronized PBroadcast<sup>4</sup> of the value  $v$  it proposes to the two-process consensus object  $CONSENSUS\{i, j\}$  (line 1). When this synchronized PBroadcast terminates,  $p_i$  decides (line 2) the proposed value that has been deposited in  $decided\{i, j\}_i$  (line 3).

**Theorem 7** Algorithm 8 solves binary consensus in the system model  $\mathcal{CAMP}_{n,t}[\text{PBroadcast}]$ .

**Proof** *Proof of the Validity property.* When process  $p_i$  returns  $decided\{i, j\}_i$  on line 2, it has already received the message PROPOSITION( $v, \{i, j\}$ ) it pbroadcast at line 1. Hence,  $p_i$  executed line 3 at least once, so  $decided\{i, j\}_i \neq \perp$ . Moreover, only proposed values can be pbroadcast within a PROPOSITION() message on line 1, so only proposed values can be assigned to  $decided\{i, j\}_i$  on line 3.

*Proof of the Agreement property.* By the Pair ordering property applied to the PROPOSITION() messages pbroadcast by  $p_i$  and  $p_j$ ,  $p_i$  and  $p_j$  receive the two messages in the same order. By line 3, they both decide the value carried by the first message they receive.

*Proof of the Termination property.* Algorithm 8 contains no loops nor recursive calls.  $\square_{\text{Theorem 7}}$

### C.2 From two-process consensus to PBroadcast

Algorithm 9 implements PBroadcast on top of consensus between two processes. It is inspired by the implementation of the total-order broadcast given in [12]. The main difference lies in the fact that the messages are not globally ordered by all processes, but rather by each pair of processes independently.

As far as shared objects are concerned, for all  $i$  and  $j \neq i$ ,  $CONSENSUS\{i, j\}$  denotes an unbounded sequence of consensus instances between  $p_i$  and  $p_j$ . The  $k^{\text{th}}$  element of this sequence is denoted  $CONSENSUS\{i, j\}[k]$ .

Moreover, each process  $p_i$  manages two local variables.

<sup>4</sup>The implementation of synchro\_pbroadcast is the same as the one of synchro\_mbroadcast described in Algorithm 1.

```

operation pb_broadcast( $m$ ) is                                     % code for  $p_i$ 
(1) reliable_broadcast PB( $m$ );
(2) for  $j$  from 1 to  $i - 1$  and then from  $i + 1$  to  $n$  do order( $m, j$ ) end for;
(3) deliver( $m, i$ )
end operation.

when PB( $m$ ) is reliable-delivered from  $p_j$ 
(4) if  $j \neq i$  then order( $m, j$ ); deliver( $m, j$ ) end if.

operation order( $m, j$ ) is
(5) repeat  $m' \leftarrow CONSENSUS\{i, j\}[ordered_i[j]].propose(m)$ ;
(6)    $ordered_i[j] \leftarrow ordered_i[j] + 1$ ;
(7)   if  $m' \neq m$  then deliver( $m', j$ ) end if
(8) until  $m' = m$ 
end operation.

operation deliver( $m, j$ ) is
(9) if  $m \notin delivered_i$  then
(10)    $delivered_i \leftarrow delivered_i \cup \{m\}$ ;
(11)   pb_delivery of  $m$  from  $p_j$ 
(12) end if
end operation.

```

Algorithm 9: PBroadcast on top of  $\mathcal{CAMP}_{n,t}[\text{consensus}_2]$

- $delivered_i$  is the set of all messages that  $p_i$  has already pdelivered (initially  $\emptyset$ ).
- $ordered_i$ , an array of  $n$  integer values (initially 0,  $ordered_i[i]$  is not used), such that  $ordered_i[j]$  is the number of consensus instances between  $p_i$  and  $p_j$  in which  $p_i$  took part, i.e. the index of the next consensus object  $CONSENSUS\{i, j\}$  that can be used by  $p_i$ .

Each pair of processes  $(p_i, p_j)$  agrees on a sequence of all messages pbroadcast by  $p_i$  or  $p_j$ , thanks to the sequence of consensus instances saved in  $CONSENSUS\{i, j\}$  (Lines 5-6). Notice that it is possible that the same message happens twice in this sequence, in which case only the first occurrence will be considered in the order in which messages are delivered (Lines 9-12).

In order to pbroadcast a message  $m$ , a process  $p_i$  first broadcasts a message PB( $m$ ) on line 1 to ensure that  $m$  will eventually win a consensus against messages from  $p_j$ . Upon delivery of this message,  $p_j$  helps  $p_i$  to win a consensus by proposing  $m$  as well on  $CONSENSUS\{i, j\}$  until some consensus is won by  $m$ . It is assumed that there is no concurrency between the execution of pb\_broadcast( $m$ ) and the code associated to the reliable delivery of PB( $m$ ).

Then  $p_i$  tries to insert its message  $m$  in the sequences it shares with all the other processes, until  $m$  is the next message it has agreed to pdeliver with all other processes (line 2), and then pdelivers  $m$  (line 3).

**Theorem 8** Algorithm 9 implements reliable-PBroadcast.

**Proof** *Proof of the Validity property.* Process  $p_i$  can only pdeliver a message  $m$  from  $p_j$  on line 11, by calling deliver( $m, j$ ). This can happen on three places. On line 3,  $p_i$  did pbroadcast  $m$ . On line 4,  $p_i$  reliable-delivered a message PB( $m$ ) from  $p_j$ , that  $p_j$  reliable-broadcast on line 1 after pbroadcasting  $m$ . The remaining case is the call to deliver( $m', j$ ) on line 7. By validity of Consensus on line 5,  $m'$  was proposed when  $p_i$  or  $p_j$  called order( $m', j$ ) or order( $m', i$ ), which indicates that  $m'$  was pbroadcast by  $p_i$  or  $p_j$ . Let us remark that  $p_i$  cannot pdeliver its own messages on line 7: if  $p_i$  already delivered  $m'$  from itself, then  $p_i$  will be filtered out by Lines 9-10; if  $p_i$  has not yet delivered  $m'$  from itself, then  $p_i$  is actually executing order( $m, j$ ) from line 2, and  $m = m'$ . Hence,  $m'$  was pbroadcast by  $p_j$  which concludes the proof of the Validity property.

*Proof of the No-duplication property.* Process  $p_i$  can only pdeliver a message  $m$  on line 11 if it was not previously in set  $delivered_i$  (line 9), and then  $m$  is added to  $delivered_i$  on line 10.

*Proof of the Local termination property.* Suppose that a correct process  $p_i$  does not terminate its invocation of  $\text{pb\_broadcast}(m_i)$ . As  $\text{pb\_broadcast}(m_i)$  itself does not contain loops or recursive calls, it means there exists  $j \neq i$  such that the invocation of  $\text{order}(m_i, j)$  does not terminate, i.e. that for all  $k$ ,  $\text{CONSENSUS}\{i, j\}[k].\text{propose}(m_i)$  never yields  $m_i$  on line 5. This is not possible since, by the Global-CF property of reliable-broadcast,  $p_j$  eventually reliable-delivers the message  $\text{PB}(m)$  that  $p_i$  reliable-broadcasts on line 1. This results in both processes proposing  $m_i$ , which must be decided by the validity property of consensus.

*Proof of the Global-CF property.* Suppose a process  $p_j$  pdelivers a message  $m_i$  from  $p_i$ . By the proof of validity above, it means that  $p_i$  pbroadcast  $m_i$ , and reliable-broadcast  $\text{PB}(m_i)$  on line 1, since  $m_i$  could have never been delivered if  $p_i$  crashed before line 1. Let  $p_k$  be a correct process. By the Global-CF property of reliable-broadcast,  $p_k$  reliable-delivered  $\text{PB}(m_i)$ . Then, by the same reasoning as developed in the proof of the Local termination property,  $p_k$  pdelivers  $m_i$  from  $p_i$ .

*Proof of the Pair ordering property.* Let  $m_i$  and  $m_j$  be two messages pbroadcast respectively by  $p_i$  and  $p_j$ . As seen above, there is  $k_i$  and  $k_j$  such that  $\text{CONSENSUS}\{i, j\}[k_i]$  decides  $m_i$  and  $\text{CONSENSUS}\{i, j\}[k_j]$  decides  $m_j$ . Let us consider the smallest such  $k_i$  and  $k_j$ , and without loss of generality, let us suppose that  $k_i < k_j$ . Remark that the pdelivery of  $m_i$  by  $p_i$  happens when  $p_i$  calls  $\text{deliver}(m_i, i)$  on line 3. At that point, its variable  $\text{ordered}_i[j]$  was last modified when it called  $\text{order}(m_i, j)$  on line 2, which returned when  $m' = m_i$ , i.e. when  $\text{ordered}_i[j] = k_i + 1$  (by Lines 5-6). Differently,  $p_i$  pdelivers  $m_j$  by calling  $\text{deliver}(m_j, j)$  either on line 4 or line 7, in both cases just after it has executed Lines 5-6, so  $\text{ordered}_i[j] = k_j + 1$ . Hence,  $p_i$  pdelivers  $m_i$  before  $m_j$ . Similarly,  $p_j$  pdelivers  $m_i$  when  $\text{ordered}_j[i] = k_i + 1$  and  $m_j$  when  $\text{ordered}_j[i] = k_j + 1$ , hence  $p_j$  pdelivers  $m_i$  before  $m_j$ . Therefore,  $p_i$  and  $p_j$  pdeliver  $m_i$  and  $m_j$  in the same order which concludes the proof of the Pair ordering property.  $\square_{\text{Theorem 8}}$

### C.3 From PBroadcast to Test&Set()

For simplicity, this section considers one-shot Test&Set(). It can be easily generalized to multi-shot Test&Set().

**Definition of Test&Set().** A test&set object is an object that can take only two values `true` or `false`. Its initial value is `true`. It provides the processes with a single one-shot atomic operation denoted  $\text{Test\&Set}()$ , that sets the value of the object to `false` and returns the previous value of the object. As the operation is atomic, its executions can be linearized and the only invocation that returns `true` is the first that appears in the linearization order. From a computability point of view the consensus number of  $\text{Test\&Set}()$  is 2 [3, 25].

**A PBroadcast-based implementation of Test&Set() and its proof.** Algorithm 10 is a PBroadcast-based implementation of a  $\text{Test\&Set}()$  object. It uses a series of two-process tournaments to elect one and only one winner. To this end each process  $p_i$  manages two local variables.

- $\text{round}_i$  is an integer between 0 and  $\max(2, \lceil \log_2(n) \rceil + 1)$  (initially 0) that represents  $p_i$ 's current progress.
- $\text{vying}_i$  a Boolean, initially `true`, that becomes `false` after  $p_i$  has lost a tournament.

The levels of the tournament tree are associated with rounds such that at each round  $r$ , two winners from round  $r - 1$  compete by PBroadcasting a message  $\text{COMPETE}(r)$  (line 2). Since both processes pdeliver both messages in the same order, the first message decides which process reaches round  $r + 1$ . More precisely, the tournament tree is such that the set of processes is partitioned into subsets of size  $2^r$ ,

```

operation Test&Set() is                                     % code for  $p_i$ 
(1) while  $round_i < 2 \vee (round_i \leq \lceil \log_2(n) \rceil \wedge vying_i)$  do
(2)   synchro_pbroadcast COMPETE( $round_i$ )
(3) end while;
(4) return( $vying_i$ )
end operation.

when COMPETE( $r$ ) is pdelivered from  $p_j$  do
(5) if  $i = j$  then  $round_i \leftarrow r + 1$ 
(6) else if  $round_i = 0 \wedge r = 1$  then  $vying_i \leftarrow \text{false}$ 
(7) else if  $round_i < r \wedge \lfloor \frac{i-1}{2^r} \rfloor = \lfloor \frac{j-1}{2^r} \rfloor$  then  $vying_i \leftarrow \text{false}$ 
(8) end if.

```

Algorithm 10: Test&Set() on top of  $\mathcal{CAM}\mathcal{P}_{n,t}[\text{PBroadcast}]$

i.e.  $\{p_1, \dots, p_{2^r}\}, \{p_{2^r+1}, \dots, p_{2 \times 2^r}\}, \dots, \{p_{\lfloor \frac{n-1}{2^r} \rfloor \times 2^r + 1}, \dots, p_n\}$ . Hence,  $p_i$  loses round  $r$  if it receives a message COMPETE( $r$ ) from another process  $p_j$  playing in the same set at round  $r$ , i.e. such that  $\lfloor \frac{i-1}{2^r} \rfloor = \lfloor \frac{j-1}{2^r} \rfloor$ , when  $round_i < r$  (line 7), which indicates that  $p_i$  did not receive its own message COMPETE( $r$ ) yet (line 5). Processes play until they lose a round or they win the finale at round  $\lceil \log_2(n) \rceil$  (condition  $round_i \leq \lceil \log_2(n) \rceil \wedge vying_i$  on line 1). Then, they return the content of their variable  $vying_i$ , which can be true only for a process that won all its tournaments.

In order to ensure linearizability, Algorithm 10 adds a round 0 to prevent late processes to win if they start their execution after another process played its first tournament:  $p_i$  forfeits if it receives a message COMPETE(1) from another process before its own message COMPETE(0) (line 6). In this case,  $p_i$  still participates in round 1, so its message COMPETE(1) forces even slower processes to forfeit as well (condition  $round_i < 2$  on line 1).

**Lemma 1** *In any execution of Algorithm 10 in which some process invokes Test&Set() and no process crashes, exactly one operation returns true.*

**Proof** Let us say that a participating process  $p_i$  “wins at round  $r$ ” if, at some point of the execution,  $round_i = r + 1$  and  $vying_i = \text{true}$ . We prove the predicate  $P(r) \wedge Q(r)$  defined as follows, by induction on  $r \in \{0, \dots, \max(2, \lceil \log_2(n) \rceil)\}$ :

- $P(r)$ : for all  $k$ , at most one process in the subset  $\{p_{k \times 2^r + 1}, \dots, p_{(k+1) \times 2^r}\}$  wins at round  $r$ ,
- $Q(r)$ : at least one process wins at round  $r$ .

*Initialization:*  $P(0)$ . For all  $k$ ,  $\{p_{k \times 2^0 + 1}, \dots, p_{(k+1) \times 2^0}\} = \{p_{k+1}\}$  contains at most one process.

*Initialization:*  $Q(0)$ . By the condition  $round_i < 2$  on line 1, all participating processes pbroadcast messages COMPETE(0) and COMPETE(1). Let  $p_i$  be the first process that pbroadcasts a message COMPETE(1). Process  $p_i$  cannot pdeliver a message COMPETE(1) before its own message COMPETE(0), hence, it does not execute line 6. Moreover,  $p_i$  does not execute line 7 when  $r = 0$  because there is no  $j \neq i$  such that  $\lfloor \frac{i-1}{2^0} \rfloor = \lfloor \frac{j-1}{2^0} \rfloor$ . Hence,  $p_i$  wins at round 1.

*Induction:*  $P(r) \wedge Q(r) \implies P(r+1)$ . Let us consider a set  $S = \{p_{k \times 2^{r+1} + 1}, \dots, p_{(k+1) \times 2^{r+1}}\}$ , for some  $k$ . If  $r = 0$ ,  $S$  contains two processes  $q_1$  and  $q_2$ . Otherwise,  $S$  can be partitioned into two subsets  $S_1 = \{p_{2k \times 2^r + 1}, \dots, p_{(2k+1) \times 2^r}\}$  and  $S_2 = \{p_{(2k+1) \times 2^r + 1}, \dots, p_{(2k+2) \times 2^r}\}$ . By  $P(r)$ , at most one process  $q_1$  (resp.  $q_2$ ) wins in  $S_1$  (resp.  $S_2$ ) at round  $r$ . In both cases, either at most one process participates at round  $r$ , hence at most one process wins at round  $r+1$ , or  $q_1$  and  $q_2$  are the only processes in  $S$  for which the condition on line 1 is true, so they are the only processes in  $S$  to pbroadcast a message COMPETE( $r+1$ ). Let  $p_j$  be the process whose message is pdelivered first, and let  $p_i$  be the other process. When  $p_i$  pdelivers  $p_j$ 's message,  $round_i < r+1$ , because  $round_i$  can only be updated to  $r+1$  on line 5

after  $p_i$  pdelivers its own message; and  $\lfloor \frac{i-1}{2^{r+1}} \rfloor = \lfloor \frac{j-1}{2^{2^{r+1}}} \rfloor$  by definition of  $S$ . Hence,  $vying_i$  is set to `false` on line 7, and only  $p_j$  wins at round  $r + 1$  in  $S$ .

*Induction:*  $P(r) \wedge Q(r) \implies Q(r + 1)$ . By  $Q(r)$ , some process  $p_i$  wins at round  $r$ . Since  $p_i$  does not crash and the condition is true on line 1,  $p_i$  pbroadcasts a message `COMPETE( $r + 1$ )`, and by  $P(r)$  at most one other process  $p_j$  does the same thing in the same set as  $p_i$ . Without loss of generality, let us assume that  $p_i$ 's message `COMPETE( $r + 1$ )` is the first that  $p_i$  and  $p_j$  pdeliver between theirs. Then  $p_i$  wins at round  $r + 1$ .

*Conclusion.* For  $r = \max(2, \lceil \log_2(n) \rceil)$ , all processes belong to the same set  $S = \{p_1, \dots, p_n\}$ , and exactly one process  $p_i \in S$  wins at round  $r$ . Then,  $p_i$  returns `true` on line 4, and all other processes return `false`.  $\square_{\text{Lemma 1}}$

**Theorem 9** *Algorithm 10 implements a wait-free linearizable `Test&Set()` object in the system model  $\mathcal{CAM}\mathcal{P}_{n,t}[\text{PBroadcast}]$ .*

**Proof** *Proof of wait-freedom.* At each iteration of the only loop (Lines 1-3), Process  $p_i$  executes line 5 at pdelivery of the message is pbroadcasts on line 2. Hence,  $round_i$  is incremented at each iteration, but cannot go passed  $\max(2, \lceil \log_2(n) \rceil + 1)$  by line 1.

*Proof of Linearizability.* Let us consider an execution  $E$  admitted by Algorithm 10. If  $E$  is the empty execution, then it is linearizable. Otherwise, since the algorithm is wait-free and there is a finite number of processes,  $E$  is finite. Therefore, there exists a finite extension  $E'$  of  $E$  in which all processes that crashed in  $E$  terminate the execution of their operation in  $E'$ .

By Lemma 1, exactly one operation  $o$ , performed by some process  $p$ , returns `true` in  $E'$ . We define the binary relation  $\rightarrow$  between operations as  $o_1 \rightarrow o_2$  if either 1)  $o_1$  was terminated before  $o_2$  was started (denoted by  $o_1 \rightarrow_1 o_2$ ), or 2)  $o_1 = o$  (denoted by  $o_1 \rightarrow_2 o_2$ ).

Let us prove that  $\rightarrow$  is cycle-free. Indeed, suppose there is a cycle  $o_1 \rightarrow o_2 \rightarrow \dots \rightarrow o_k = o_1$  containing at least two different operations. Since  $\rightarrow_1$  is cycle-free,  $o$  is part of the cycle, so there exists  $k$  such that  $o_k \rightarrow_1 o$ , where  $o_k$  is executed by some process  $p_k$ . By the condition  $round_i < 2$  on line 1,  $p_k$  pbroadcasts a message `COMPETE(1)` on line 2. Process  $p_k$  pdelivers its message `COMPETE(1)` before it returns from  $o_k$ , which happens before  $p$  starts  $o$ . Hence,  $p$  pdelivers  $p_k$ 's message before any message it pbroadcasts during  $o$ . In particular,  $p$  returns  $vying_i$  on line 4 after setting  $vying_i$  to `false` on line 6, which contradicts the fact that  $o$  returns `true`.

Finally, the reflexive and transitive closure of  $\rightarrow$  can be extended into a total order that respects real time thanks to  $\rightarrow_1$ , and such that the first operation returns `true` and the others return `false` thanks to  $\rightarrow_2$ . Hence,  $E'$  is linearizable. In other words, all operations started in  $E$  can be completed, so that the resulting execution  $E'$  is linearizable, which concludes the proof.

$\square_{\text{Theorem 9}}$