Send/Receive Patterns Versus Read/Write Patterns in Crash-Prone Asynchronous Distributed Systems

Mathilde Déprés
École Normale Supérieure de Paris-Saclay, France

Achour Mostéfaoui
LS2N, Nantes Université, France

Matthieu Perrin
LS2N, Nantes Université, France

Michel Raynal
Univ Rennes IRISA, Inria, CNRS, 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.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Computer systems organization → Fault-tolerant network topologies; Networks → Programming interfaces

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

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.16

Related Version *Full Version*: https://hal.science/hal-04087447
Previous Version: https://doi.org/10.1145/3583668.3594569

Funding This work was partially supported by the French "Étoile Montante en Pays De La Loire" regional project BROCCOLI devoted to the computability aspects of broadcast abstractions and the French ANR project ByBloS (ANR-20-CE25-0002-01) and PriCLeSS (ANR-10-LABX-07-81).

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 [47].

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, 26, 41, 50] 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 [32]. 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., [31] 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, [28] 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[1].

Based on the same principles as ABD, algorithms building RW registers have been designed [52]. Some strive to reduce the size of control information carried by messages (e.g. [6, 42]) while others focus on fast read and/or fast write operations in good circumstances (e.g. [21, 25, 43]). 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 [16] 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.

1.4 Content of the article

In the spirit of [24] (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.
- 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

[1] 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.

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 [37]). Section 7 presents a simple version of the well-known Paxos consensus algorithm [36].[2] 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). The proofs of the algorithms are presented in an appendix.[3]

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 $p_1, ..., p_n$. 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 unidirectional 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

[2] It is worth noticing that mutex and consensus are the two most famous distributed computing problems [49]. Additionally, Appendix B presents a very simple MBroadcast-based algorithm that builds a lattice agreement object.

[3] The definition of MBroadcast was first presented in a short version at PODC 2023 [18], and an extended version is available on the web [19].

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, 26, 34, 41, 50, 47]. 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, ..., 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.

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

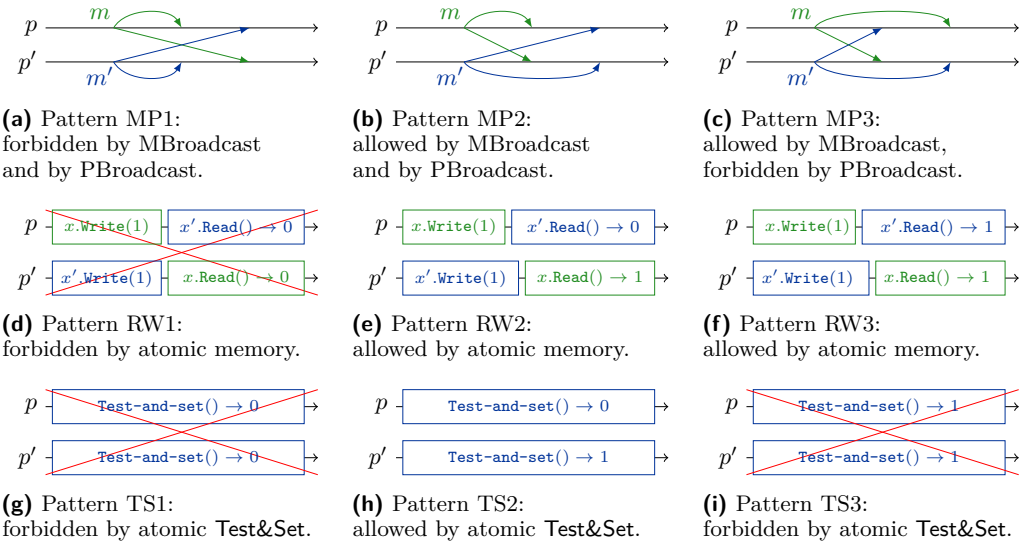
3.1 Three basic binary message patterns

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).

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).



(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: forbidden 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.

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 [29, 35] 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.

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 [51]. 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.

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.

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 mbroadcast() and mdeliver(). When a process invokes mbroadcast(m) we say "it mbroadcasts the message m". The invocation of mdeliver() 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 it 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 mbroadcast(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 mbroadcast(m), it returns from its invocation.

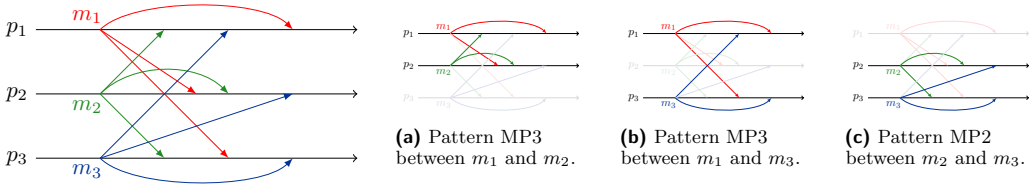
Global CS-termination. If a correct process invokes mbroadcast(m), all correct processes mdeliver m. ("CS" is used to stress the fact that the sender is required to be correct.)

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).

As it is the case with other broadcast abstractions, MBroadcast can be enriched with other properties, defined in [18], 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 that also respect the properties of fifo-broadcast, causal-broadcast and reliable-broadcast, respectively.

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.



(a) Pattern MP3 between m_1 and m_2.

(b) Pattern MP3 between m_1 and m_3.

(c) Pattern MP2 between m_2 and m_3.

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 hides the system parameter t to 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 mdeliver m' before m. This pattern is described in Figure 3. In other words, sending a "blank" synchronization message and waiting for its delivery is sufficient to "harvest" all the messages that were already mdelivered by their senders. To benefit from this property, we consider the synchronized broadcast operation based on a synchronization barrier as defined in Algorithm 1.

Algorithm 1 Synchronized MBroadcast.

```
operation synchro_mbroadcast(m) is
                                   % code for p_i
(1)  mbroadcast(m);
(2)  wait (m has been mdelivered from p_i)
end operation.
```

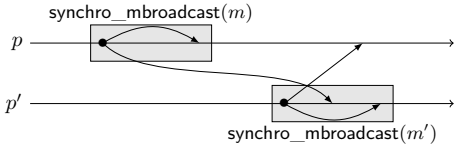


Figure 3 A real-time property.

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 mdelivered m.*

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 pbroadcast() and pdeliver(). When a process invokes pbroadcast(m) we say "it pbroadcasts the message m". The invocation of pdeliver() 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 it 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 [27] (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 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 computationnaly 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 occuring makes MBroadcast equivalent to shared memory that prevents the memory pattern RW1.

Algorithm 2 MBroadcast on top of $\mathcal{CARW}_{n,t}[\emptyset]$.

```
operation mbroadcast(m) is                                   % code for p_i
(1)  SENT[i] ← SENT[i] ⊕ m;
(2)  catch_up();
(3)  mdelivery 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 ← SENT[j];
(6)    for k from delivered_i[j] to |msgj| − 1 do mdelivery of msgj[k + 1] from p_j end for;
(7)    delivered_i[j] ← |msgj|
(8)  end for.
```

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, ..., p_n broadcasting respectively messages m_1, ..., 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_ℓ). 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 ∞.

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, so they can cope with any number of process crashes.

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 [35].

Shared memory and local variables.

- The n processes share an array of n SWMR regular registers denoted $SENT[1..n]$ such that, for any i, $SENT[i]$ 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 $SENT[i]$ will be in position 1, the second in position 2, etc.
- 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).

Algorithm 3 Atomic RW register on top of $\mathcal{CAMP}_{n,t}$[MBroadcast].

```
operation write(v) is                                   % code for p_i
(1)  synchro_mbroadcast SYNCH();
(2)  let t_i such that ⟨t_i, i⟩ > clock_i;
(3)  synchro_mbroadcast WRITE(v, ⟨t_i, i⟩)
end operation.

operation read() is
(4)  synchro_mbroadcast SYNCH();
(5)  v ← 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 ← v; clock_i ← c end if.
```

Description of the algorithm. Algorithm 2 builds reliable-MBroadcast on top of n regular RW registers. When a process p_i invokes mbroadcast(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* 2 *builds* MBroadcast *on top of* regular RW registers. (Proof in Appendix A.)

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 MWMR 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 3 implements an MWMR atomic register on top of MBroadcast, i.e., in the model $\mathcal{CAMP}_{n,t}$[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, 39, 46]. 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* 3 *implements an* atomic RW register *on top of* $\mathcal{CAMP}_{n,t}$[MBroadcast]. (Proof in Appendix A.)

Suppressing Line 1 (resp. Line 6) builds an SWMR atomic register (resp. an SWMR regular register). Algorithm 3 can also be easily adapted to work with the four types of MWMR regular registers defined in [53].

5.3 A remark on complexity

Although Algorithms 2 and 3 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 2 to implement MBroadcast.

In fact, this complexity is necessary, even with the use of multi-writer multi-reader atomic registers. It was proven in [30], 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 2 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.

Conversely, it is possible to implement the MBroadcast abstraction in the system model $\mathcal{CAMP}_{n,t}[t < n/2]$ ($\mathcal{CAMP}_{n,t}[\emptyset]$ enriched with the constraint $t < n/2$), using the algorithm presented in [18], whose cost is only $O(n)$ implementation messages when mutual broadcast delivery concerns only correct processes.

5.4 What is actually needed to build a RW register

It follows from the previous results that the operational condition

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

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 MP2 $\vee$ 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 MP2 $\vee$ 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 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 3), in any message-passing system where consensus is 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 [35], but, to our knowledge, it had never been stated explicitly.

Algorithm 4 An MBroadcast-based version of Lamport's Bakery algorithm.

```
operation acquire() is                          % code for p_i
(1)  fifo_synchro_mbroadcast HELLO();
(2)  let t_i such that ⟨t_i, i⟩ > max(tickets_i);
(3)  fifo_synchro_mbroadcast TICKET(t_i);
(4)  wait(⟨t_i, i⟩ = 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 ← ticket_i ∪ ⟨0, j⟩.

when TICKET(t) is fifo-mdelivered from p_j do
(7)  ticket_i ← (ticket_i \ {⟨0, j⟩}) ∪ {⟨t, j⟩}.

when GOODBYE(t) is fifo-delivered from p_j do
(8)  ticket_i ← ticket_i \ {⟨t, j⟩}.
```

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 explicitely based on quorums. Moreover, as the reader will see, the mutex algorithm and the consensus algorithm have the same structure as Algorithm 3.

6.1 Mutex

Considering the asynchronous message-passing system (i.e. the computing model $\mathcal{CAMP}_{n,t}[\emptyset]$) this section addresses the mutual exclusion problem (mutex). This problem was introduced by E.W. Dijkstra in 1965 [20]. From a historical point of view it is the first distributed computing problem (see [51] 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 [33, 37]. 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 4 is an MBroadcast-based version of Lamport's Bakery algorithm [33]. 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 $\mathcal{CAMP}_{n,t}$[MBroadcast], Algorithm 4 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 in Appendix A.)

Remark. Using the failure detector introduced in [17], 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 [38, 44] 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 [23] or RW registers [40]. We consider here that this additional computability power

Algorithm 5 An MBroadcast-based variant of Lamport's Paxos algorithm.

```
operation propose(v_i) is                                          % code for p_i
(1)   val_i ← v_i;
(2)   while (decided_i = ⊥) do
(3)       if (leader() = i) then
(4)           let t_i such that ⟨t_i, i⟩ > clock_i;
(5)           synchro_mbroadcast BEGIN(t_i);
(6)           if ⟨t_i, i⟩ = clock_i then synchro_mbroadcast VOTED(t, val_i) end if;
(7)           if ⟨t_i, i⟩ = 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 ⟨t, j⟩ > clock_i then clock_i ← ⟨t, j⟩ end if.

when VOTED(t, v) is mdelivered from p_j do
(12)  if ⟨t, j⟩ > clock_i then clock_i ← ⟨t, j⟩; val_i ← v end if.

when SUCCESS(v) is reliable-delivered from p_j do
(13)  decided_i ← v.
```

is given by the failure detector denoted Ω, which is the weakest failure detector with which consensus can be solved [11]. Ω 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

Local variables. Each process p_i manages three local variables.

- $decided_i$, initialized to $\perp$ (a default value that cannot be proposed to consensus), 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.

Description of the algorithm. Algorithm 5 solves the consensus problem in the system model $\mathcal{CAMP}_{n,t}[\text{MBroadcast}, \Omega]$. When a process p_i invokes 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 Ω (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 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 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* 5 *solves* consensus *in* $\mathcal{CAMP}_{n,t}$[MBroadcast,Ω]. (Proof in [19].)

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 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, 27, 45, 48]. 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 ∞ 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.[4]

References

1 Yehuda Afek, Danny Dolev, Hagit Attiya, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. In *Proc. of the 9th Annual ACM Symposium on Principles of Distributed Computing, Quebec, Canada, August 22-24, 1990*, pages 1–13, 1990.

2 Yehuda Afek, Faith Ellen, and Eli Gafni. Deterministic objects: Life beyond consensus. In *Proc. of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*, pages 97–106, 2016.

[4] Let us remind that, it has been shown in [15] that the weakest failure detector to implement Test&Set and Compare&Swap in asynchronous shared-memory systems prone to any number of crashes is the same failure detector (namely, Ω).

3 Yehuda Afek, Eytan Weisberger, and Hanan Weisman. A completeness theorem for a class of synchronization objects (extended abstract). In *Proc. of the Twelth Annual ACM Symposium on Principles of Distributed Computing, Ithaca, New York, USA, August 15-18, 1993*, pages 159–170, 1993.

4 Mustaque Ahamad, Gil Neiger, James E Burns, Prince Kohli, and Phillip W Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

5 James H. Anderson. Multi-writer composite registers. *Distributed Comput.*, 7(4):175–195, 1994.

6 Hagit Attiya. Efficient and robust sharing of memory in message-passing systems. *J. Algorithms*, 34(1):109–127, 2000.

7 Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, 1995.

8 Hagit Attiya, Maurice Herlihy, and Ophir Rachman. Atomic snapshots using lattice agreement. *Distributed Computing*, 8(3):121–132, 1995.

9 Hagit Attiya and Jennifer L. Welch. *Distributed computing - fundamentals, simulations, and advanced topics (2. ed.)*. Wiley series on parallel and distributed computing. Wiley, 2004.

10 Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, 1987.

11 Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, 1996.

12 Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

13 Eli Daian, Giuliano Losa, Yehuda Afek, and Eli Gafni. A wealth of sub-consensus deterministic objects. In *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018*, volume 121 of *LIPIcs*, pages 17:1–17:17, 2018.

14 Luciano Freitas de Souza, Petr Kuznetsov, Thibault Rieutord, and Sara Tucci Piergiovanni. Accountability and reconfiguration: Self-healing lattice agreement. In *25th International Conference on Principles of Distributed Systems, OPODIS 2021, December 13-15, 2021, Strasbourg, France*, volume 217 of *LIPIcs*, pages 25:1–25:23, 2021.

15 Carole Delporte-Gallet, Hugues Fauconnier, and Rachid Guerraoui. Tight failure detection bounds on atomic object implementations. *J. ACM*, 57(4):22:1–22:32, 2010.

16 Carole Delporte-Gallet, Hugues Fauconnier, Sergio Rajsbaum, and Michel Raynal. Implementing snapshot objects on top of crash-prone asynchronous message-passing systems. *IEEE Trans. Parallel Distributed Syst.*, 29(9):2033–2045, 2018.

17 Carole Delporte-Gallet, Hugues Fauconnier, and Michel Raynal. On the weakest information on failures to solve mutual exclusion and consensus in asynchronous crash-prone read/write systems. *J. Parallel Distributed Comput.*, 153:110–118, 2021.

18 Mathilde Déprés, Achour Mostéfaoui, Matthieu Perrin, and Michel Raynal. Brief announcement: The mbroadcast abstraction. In *Proc. of the 2023 ACM Symposium on Principles of Distributed Computing, PODC 2023, Orlando, FL, USA, June 19-23, 2023*, pages 282–285, 2023.

19 Mathilde Déprés, Achour Mostéfaoui, Matthieu Perrin, and Michel Raynal. Send/Receive Patterns versus Read/Write Patterns: the MB-Broadcast Abstraction (Extended Version). Research report, University of Nantes, 2023. URL: `https://hal.archives-ouvertes.fr/hal-04087447`.

20 Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, 1965.

21 Partha Dutta, Rachid Guerraoui, Ron R. Levy, and Marko Vukolic. Fast access to distributed atomic memory. *SIAM J. Comput.*, 39(8):3752–3783, 2010.

22 Jose M. Faleiro, Sriram K. Rajamani, Kaushik Rajan, G. Ramalingam, and Kapil Vaswani. Generalized lattice agreement. In Darek Kowalski and Alessandro Panconesi, editors, *ACM*

Symposium on Principles of Distributed Computing, PODC '12, Funchal, Madeira, Portugal, July 16-18, 2012, pages 125–134. ACM, 2012.

23 Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

24 Michael J. Fischer and Michael Merritt. Appraising two decades of distributed computing theory research. *Distributed Comput.*, 16(2-3):239–247, 2003.

25 Chryssis Georgiou, Theophanis Hadjistasi, Nicolas Nicolaou, and Alexander A. Schwarzmann. Implementing three exchange read operations for distributed atomic storage. *J. Parallel Distributed Comput.*, 163:97–113, 2022.

26 Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report Tech Report 94-1425, Cornell University, 1994. Extended version of "Fault-Tolerant Broadcasts and Related Problems" in *Distributed systems, 2nd Edition*, Addison-Wesley/ACM, pp. 97-145 (1993.

27 Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

28 Maurice Herlihy, Sergio Rajsbaum, Michel Raynal, and Julien Stainer. From wait-free to arbitrary concurrent solo executions in colorless distributed computing. *Theor. Comput. Sci.*, 683:1–21, 2017.

29 Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

30 Damien Imbs, Petr Kuznetsov, and Thibault Rieutord. Progress-space tradeoffs in single-writer memory implementations. In *21st International Conference on Principles of Distributed Systems, OPODIS 2017, Lisbon, Portugal, December 18-20, 2017*, volume 95 of *LIPIcs*, pages 9:1–9:17, 2017.

31 Damien Imbs, Achour Mostéfaoui, Matthieu Perrin, and Michel Raynal. Which broadcast abstraction captures k-set agreement? In Andréa W. Richa, editor, *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, volume 91 of *LIPIcs*, pages 27:1–27:16, 2017.

32 Damien Imbs, Achour Mostéfaoui, Matthieu Perrin, and Michel Raynal. Set-constrained delivery broadcast: A communication abstraction for read/write implementable distributed objects. *Theor. Comput. Sci.*, 886:49–68, 2021.

33 Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.

34 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

35 Leslie Lamport. On interprocess communication. part I: basic formalism. *Distributed Comput.*, 1(2):77–85, 1986.

36 Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

37 Leslie Lamport. Deconstructing the bakery to build a distributed state machine. *Commun. ACM*, 65(9):58–66, 2022.

38 Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

39 Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

40 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.

41 Anshuman Misra and Ajay D. Kshemkalyani. Solvability of byzantine fault-tolerant causal ordering problems. In Mohammed-Amine Koulali and Mira Mezini, editors, *Proc. of the 10th International Conference on Networked Systems, NETYS 2022, Virtual Event*, volume 13464 of *Lecture Notes in Computer Science*, pages 87–103. Springer, 2022.

42 Achour Mostéfaoui and Michel Raynal. Two-bit messages are sufficient to implement atomic read/write registers in crash-prone systems. In George Giakkoupis, editor, *Proc. of the 2016*

ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016, pages 381–389. ACM, 2016.

43 Achour Mostéfaoui, Michel Raynal, and Matthieu Roy. Time-efficient read/write register in crash-prone asynchronous message-passing systems. *Computing*, 101(1):3–17, 2019.

44 Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

45 Matthieu Perrin, Achour Mostéfaoui, Grégoire Bonin, and Ludmila Courtillat-Piazza. Extending the wait-free hierarchy to multi-threaded systems. *Distributed Computing*, 35(4):375–398, 2022.

46 Michel Raynal. *Concurrent Programming - Algorithms, Principles, and Foundations*. Springer, 2013.

47 Michel Raynal. *Fault-Tolerant Message-Passing Distributed Systems - An Algorithmic Approach*. Springer, 2018.

48 Michel Raynal. *Concurrent crash-prone shared memory systems: a few theoretical notions*. Morgan & Claypool Publishers, 2022.

49 Michel Raynal. Mutual exclusion vs consensus: both sides of the same coin? *Bull. EATCS*, 140, 2023.

50 Michel Raynal, André Schiper, and Sam Toueg. The causal ordering abstraction and a simple way to implement it. *Information Process. Letters*, 39(6):343–350, 1991.

51 Michel Raynal and Gadi Taubenfeld. A visit to mutual exclusion in seven dates. *Theor. Comput. Sci.*, 919:47–65, 2022.

52 Eric Ruppert. Implementing shared registers in asynchronous message-passing systems. In *Encyclopedia of Algorithms - 2008 Edition*. Springer, 2008.

53 Cheng Shao, Jennifer L. Welch, Evelyn Pierce, and Hyunyoung Lee. Multiwriter consistency conditions for shared memory registers. *SIAM J. Comput.*, 40(1):28–62, 2011.

A Missing proofs

A.1 From atomic RW registers to MBroadcast

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

Proof. Let us consider an execution of Algorithm 2. 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^{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 2 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. ◀

A.2 From MBroadcast to atomic RW registers

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

Proof. Let us first remark that Algorithm 3 contains no loop, so all its operations terminate.

Let us consider an execution admitted by Algorithm 3. 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 $o_1 \rightarrow o_2 \rightarrow \ldots \rightarrow o_k = o_1$ 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. ◀

A.3 Proof of the Mutex algorithm

▶ **Theorem 4.** *Considering the system model* $\mathcal{CAMP}_{n,t}$[MBroadcast]*, Algorithm* 4 *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 $\langle t, i\rangle = \mathsf{min}(ticket_i)$. A contradiction. ◀

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 [22]. More recently, lattice agreement has been used as a building block to solve accountability and reconfiguration issues encountered in distributed computing [14].

B.1 Definition

A bounded join-semilattice $(L, \bot, \sqsubseteq, \sqcup)$ is composed of a set L of elements partially ordered according to a relation $\sqsubseteq$, such that there is a smallest element $\bot$ 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.

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.

Algorithm 6 Lattice agreement in $\mathcal{CAMP}_{n,t}$[MBroadcast].

```
operation propose(v_i) is                    % code for p_i
(1)  repeat prev_i ← view_i;
(2)         synchro_mbroadcast STATE(view_i ⊔ 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 ← view_i ⊔ v.
```

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 $\bot$.
- $prev_i$ is an auxiliary variable.

Description of the algorithm. Algorithm 6 solves lattice agreement in the system model $\mathcal{CAMP}_{n,t}$[MBroadcast]. It is based on the classical double-scan principle. When a process p_i invokes 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 propose(v_i).

▶ **Theorem 6.** *Algorithm 6 solves lattice agreement in the system model* $\mathcal{CAMP}_{n,t}$[MBroadcast]. (Proof in [19].)

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

C.1 From two-process consensus to PBroadcast

Algorithm 7 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^{th} element of this sequence is denoted $CONSENSUS\{i,j\}[k]$.

Moreover, each process p_i manages two local variables.

- $delivered_i$ 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 pbroadcast(m) and the code associated to the reliable delivery of PB(m).

Algorithm 7 PBroadcast on top of $\mathcal{CAMP}_{n,t}[\text{consensus}_2]$.

```
operation pbroadcast(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 ≠ i then order(m, j); deliver(m, j) end if.

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

operation deliver(m, j) is
(9)  if m ∉ delivered_i then
(10)         delivered_i ← delivered_i ∪ {m};
(11)         pdelivery of m from p_j
(12) end if
end operation.
```

Then p_i tries to insert its message m in the sequences it shares will 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 7.** *Algorithm 7 implements* reliable-PBroadcast*.* (Proof in [19].)

C.2 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 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 Test&Set() is 2 [3, 27].

A PBroadcast-based implementation of Test&Set() and its proof. Algorithm 8 is a PBroadcast-based implementation of a 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.

- $round_i$ is an integer between 0 and $\max(2, \lceil \log_2(n) \rceil + 1)$ (initially 0) that represents p_i's current progress.
- $vying_i$ 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 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, i.e. $\{p_1, ..., p_{2^r}\}$, $\{p_{2^r+1}, ..., p_{2\times 2^r}\}$, ...,

Algorithm 8 Test&Set() on top of $\mathcal{CAMP}_{n,t}$[PBroadcast].

```
operation Test&Set() is                                   % code for p_i
(1)  while round_i < 2 ∨ (round_i ≤ ⌈log_2(n)⌉ ∧ 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 ← r + 1
(6)  else if round_i = 0 ∧ r = 1 then vying_i ← false
(7)  else if round_i < r ∧ ⌊(i−1)/2^r⌋ = ⌊(j−1)/2^r⌋ then vying_i ← false
(8)  end if.
```

$\left\{p_{\lfloor \frac{n-1}{2^r} \rfloor \times 2^r+1}, ..., p_n\right\}$. Hence, p_i looses 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 loose 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 8 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).

▶ **Theorem 8.** *Algorithm 8 implements a wait-free linearizable* Test&Set() *object in the system model* $\mathcal{CAMP}_{n,t}$[PBroadcast]. (Proof in [19].)