A Short Visit to Distributed Computing Where Simplicity is Considered a First Class Property

Michel Raynal

Institut Universitaire de France
& IRISA, Université de Rennes, France
raynal@irisa.fr

Abstract. Similarly to the injunction "*Know yourself*" engraved on the frontispiece of Delphi's temple more than two millennia ago, the sentence "*Make it as simple as possible, but not simpler*" (attributed to Einstein) should be engraved on the entrance door of all research laboratories. At the long run, what does remain of our work? Mathematicians and physicists have formulas. We have algorithms! The main issue with simplicity is that it is often confused with triviality, but as stated by J. Perlis, the recipient of the first Turing Award, "Simplicity does not precede complexity, but follows it".

Considering my research domain, namely distributed computing, this article surveys topics I was interested in during my career and presents a few results, approaches, and algorithms I designed (with colleagues). The design of these algorithms strove to consider (maybe unsuccessfully) concision, generality, simplicity, and elegance as first class properties. Said in other words, this article does not claim objectivity.

Keywords: Anonymity, Agreement, Asynchronous system, Byzantine process, Causality, Consensus, Crash failure, Distributed algorithm, Fault-tolerance, Impossibility, Message-passing, Read/write register.

1 Introduction

This article describes some of my past works in distributed computing in which generality and simplicity where considered as first class citizens. These works concern mainly causality and fault-tolerant agreement, topics that I consider as belonging to the most fundamental topics of distributed computing.

Before entering the scientific part, the section that follows shortly presents my view of my job (see [63] for an expanded view in French). Even if we disagree, it is important for each of us to have a *clear view* of what is her/his job. This can help us not only in making appropriate decisions, but also in providing us with a clearer motivation to do what we have to do!

1.1 University professor: what does it mean? A personal view

The job of a University Professor combines research and teaching activities, which are the two faces of a same coin. On the teaching side we are in the domain of "organized truths" established and sieved by our predecessors and recognized by the community as being the important knowledge that students have to master to be able to benefit from it to understand, face, and solve the problems they will encountered in their professional life [65]. In few words "*teaching is not an accumulation of facts*" [42].

On the research side, we are in a domain of "uncertainty" where we do not always know which are the most challenging and important topics among the many possible paths, trying to progress to news ideas with backtracking, hesitations and faith in what we are doing. It is important to never forget that "*it is not by improving the candle technology that electricity has been discovered and understood and its applications have been mastered*".

Let us remark that, when looking at the curricula of master degrees, the frontier between "organized truths" and "uncertainty" becomes permeable [65], which is good news. The knowledge and the questioning of students create new ideas and new approaches that entail an "industrial revolution" when they are hired by companies. This is our main impact on the society. Our job is also to ensure that today's students will still have a job in twenty years! The pair teaching/research is the key to attain this goal.

1.2 Content

As already said, this article describes some of my past works in distributed computing in which generality and simplicity where considered as first class citizens. These works concern mainly causality and agreement problems, topics which are at the core of distributed computing.

1.3 Distributed computing

Since their first instances found in field-area or interest-rate computations at the Babylonian times, algorithms have a very long history (see e.g., [38,39,56]), and later, thanks to computing machinery [8,26,29,30], they constitute the heart of *informatics*[1].

Distributed computing was born in the late 1970s when researchers and practitioners started taking into account the intrinsic characteristic of physically distributed systems. The field then emerged as a specialized research area distinct from networking, operating systems, and parallel computing.

[1] I do not like the words "computer science" and (as in a lot of European countries) I use the word "informatics" [71]. This is not only to mimic the terms "mathematics", "physics", "chemistry", etc., but is due to the fact that when we use several words to capture a scientific domain, the words we use are not equal. We do not have to confuse the name of a scientific domain (single term) and a part of its definition. As for concepts: a scientific domain, a term.

Distributed computing arises when one has to solve a problem in terms of predefined distributed entities (usually called processors, nodes, processes, actors, agents, sensors, peers, etc.) such that each entity has only a partial knowledge of the many parameters involved in the problem that has to be solved. The computing entities are pre-defined and imposed to the distributed system programmer, whose job consists in designing and implementing distributed abstractions (communication abstractions, agreement abstractions, memory abstractions, cooperation abstractions, etc.) that will be used by upper layer distributed applications. While parallel computing and real-time computing can be characterized, respectively, by the terms *efficiency* and *on-time computing*, distributed computing can be characterized by the term *uncertainty*[2]. This uncertainty is created by asynchrony, multiplicity of control flows, absence of shared memory and global time, failure, dynamicity, mobility, etc. Mastering one form or another of uncertainty is pervasive in all distributed computing problems. A main difficulty in designing distributed algorithms comes from the fact that no entity cooperating in the achievement of a common goal can have an instantaneous knowledge of the current state of the other entities, it can only know some of their past local states.

Although distributed algorithms are often made up of a few lines, their behavior can be difficult to understand and their properties hard to state and prove. Hence, distributed computing is not only a fundamental topic but also a challenging topic where simplicity, elegance, and beauty are first-class citizens [3,21].

2 Causal Message Delivery

2.1 A causality-related problem

Let us consider consider a reliable asynchronous message-passing system made up of n processes p_1, ..., p_n, where i the identity of p_i. *Causal message delivery* was introduced by K.P. Birman and T.A. Joseph in [10]. This notion, much debated when it was proposed [11], is now well-established and used in many distributed systems. Based on Lamport's happened-before relation [40], it requires that, if the sending of two messages m and m' are causally dependent, m be delivered before m'. In the context where each message is broadcast to all processes, this means that no process deliver m' before m.

A simple example is described in Fig. 1. As the sending of the messages m_1 and m_2 are independent, these messages may be delivered in different order at distinct processes. Differently, m_3 depends on m_1 but does not depend on m_2. So, m_1 must be delivered before m_3 at any process, while m_2 and m_3 can be delivered in different order at different processes. If follows that the execution

[2] In distributed computing, the computing entities are imposed to the programmer who has to allow them to correctly cooperate [64]. In parallel computing, the definition and the creation of the computing entities are under the control of the programmer: the main issue is to benefit from data independence to accordingly decompose a problem in independent sub-problems.

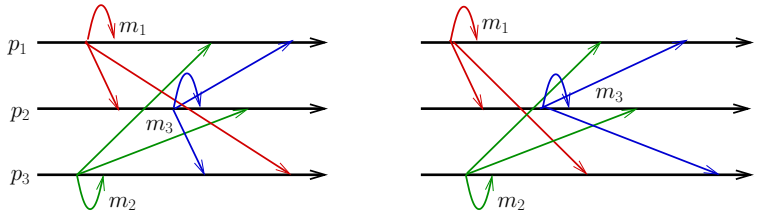

Fig. 1. Illustration of causal broadcast

depicted on the right side respects causal message delivery while the one on the left side does not (process p_3 delivers m_3 before m_1).

2.2 A very simple algorithm

The first algorithm implementing causal message delivery required each message to piggyback all the messages belonging to it causal past [10]. Differently the very simple algorithm proposed in [67] (joint work with by A. Schiper and S. Toueg) described in Fig. 2, requires each message to carry a digest of it causal past (captured with an array of integers with one entry per process).

The operations offered to the upper application layer are denoted co_broadcast and co_delivery.

```
operation co_broadcast(m) is                                   (code for p_i)
(01)  for each j ∈ {1,...,n} \ {i} do send (m, broadcast_i[1..n]) to p_j end for;
(02)  broadcast_i[i] ← broadcast_i[i] + 1;
(03)  local co_delivery of m to the application layer.

when (m, broadcast[1..n]) is received from p_j do
(04)  wait (∀k : broadcast_i[k] ≥ broadcast[k]);
(05)  local co_delivery of m to the application layer;
(06)  broadcast_i[j] ← broadcast_i[j] + 1.
```

Fig. 2. Causal message delivery broadcast [67]

Each process manages a local array, denoted $broadcast_i[1..n]$ initialized to $[0, \cdots, 0]$. The entry $broadcast_i[j]$ counts the number of messages co-broadcast by p_j as known by p_i. A process p_i "knows the co-broadcast of a message m" when it co-delivers m. The key of the algorithm is the co-delivery predicate (line 04) which states that a message become co-deliverable when all the messages belonging to its causal past have been co-delivered. Using the technique developed in [35], the size of the control information can be reduced to the minimum that is possible.

Other causal delivery algorithms are described in [62]. An efficient causal message delivery algorithm that copes with process crashes is described in [48].

3 Causality-Related Issues in Distributed Checkpointing

The notions and results presented in this section are due to joint work with R. Baldoni, J.M. Hélary, A. Mostéfaoui, and R. Netzer.

3.1 On the consistency of distributed global states

A distributed execution can be represented by a partial order on the internal/send/receive events produced by the processes [40]. This partial order generates an "equivalent" partial order on the local states of the processes produced by these events.

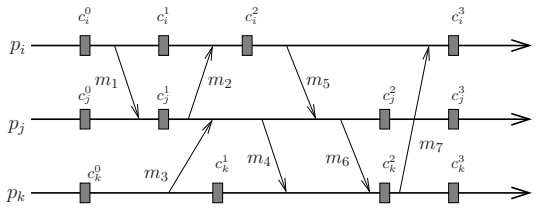

Fig. 3. Example of a distributed execution

As an example let us look at Fig. 3 that describes a simple distributed execution involving three processes p_i, p_j and p_k. As in Fig. 1, an horizontal axis is associated with each process, and small rectangles describe a subset of the process local states. The messages m_1, .., m_7 exchanged by the processes are represented by up-to-down or down-to-up arrows.

A consistent global state of the execution is made up of one local state per process, plus the messages that are in transit with respect to these local states. As an example the global state $\langle c_i^1, c_j^1, c_k^1 \rangle$ is a consistent global state in which the channels are empty except the channel from p_k to p_j which contains the message m_3. Differently the global state $\langle c_i^2, c_j^1, c_k^1 \rangle$ is not consistent (the message m_2 appears as not sent in c_j^1 and as received in c_i^2 (such a message is called an orphan message). The first algorithm that computes consistent global states was proposed by Chandy and Lamport in 1985 [18].

3.2 Z-dependency: zigzag paths and z-cycles

Definitions As shown on the figure, let us assume that, independently of each other, each process saves some of its local states in order to define a global checkpoint (global state) from which the computation can be safely restarted.

In such a context the important question becomes: given a set of local states (at most one per process), do these local states belong to a consistent global checkpoint?

To answer this question, given two local states $c1$ and $c2$, let us define two notions of a path from $c1$ to $c2$. Let $p(c1)$ and $p(c2)$ be the processes that saved the local states $c1$ and $c2$, respectively.

- Causal path from $c1$ to $c2$ (introduced in [40]), denoted $c1 \stackrel{cp}{\rightarrow} c2$, if
  - $c1$ and $c2$ have been been produced by the same process with $c1$ first, or
  - there is a sequence of messages $m_1, ..., m_y$ such that (a) m_1 was sent by $p(c1)$, (b) m_y was received by $p(c2)$, and (c) for $1 < x < y$ the process that sent m_x has previously received m_{x-1}.

  We can see that $c_i^0 \stackrel{cp}{\rightarrow} c_k^2$, and $\neg(c_k^0 \stackrel{cp}{\rightarrow} c_i^2)$.
- Interval between two local states. Let interval I_i^x be the sequence of events produced by p_i between its two *consecutive* local states c_i^x and c_i^{x+1}.
- Zigzag path from $c1$ to $c2$ (introduced in [34,55]), denoted $c1 \stackrel{zz}{\rightarrow} c2$, if %
  - $c1 \stackrel{cp}{\rightarrow} c2$, or
  - there is a sequence of messages $m_1, ..., m_y$ such that (a) m_1 was sent by $p(c1)$, (b) m_y was received by $p(c2)$, and (c) for $1 < x < y$, $p(cx)$, the process that sent m_x, sent m_x and received m_{x-1} in the same interval.

  We can see that $c_k^0 \stackrel{zz}{\rightarrow} c_i^2$ (due to the fact that p_j received m_3 and sent m_2 in the same interval).
- A Z-cycle is a zigzag path from a local state to itself. We can see that, due the messages m_5 and m_7 that are sent and received by p_i in the very same interval, we have $c_k^2 \stackrel{zz}{\rightarrow} c_k^2$.

As we are about to see, the notion of a zigzag path captures hidden causality.

3.3 A few results on distributed checkpointing: two theorems

Let a *useless* local state be a local state that cannot belong to a consistent global state. This means that a distributed checkpointing algorithm must prevent a process from saving such local states.

Let us associate an integer (a local date from an operational point of view) with each local state c saved by a process. Hence, this integer is denoted $c.date$. The two following theorems are proved in [31]. Let C be the directed graph the vertices of which are the local states saved by the processes and the directed edges are defined by the zigzag paths connecting local states.

Theorem 1. $\big(\forall\, c_1, c_2 \in C : (c_1 \stackrel{zz}{\rightarrow} c_2) \Rightarrow c_1.date < c_2.date\big) \;\Leftrightarrow\; C$ *is z-cycle-free.*

This theorem shows that all the checkpointing algorithms ensuring the z-cycle-freedom property implement (in an explicit or implicit way) a consistent logical dating of the local checkpoints (the time notion being here Lamport's clocks). It follows that, when considering the algorithms that implement explicitly such a consistent dating system, the local checkpoints that have the same date belong to the same consistent global checkpoint.

Theorem 2. *Let us assume that the initial state of each process is a local checkpoint dated* 0. *Let C be a z-cycle-free set of local checkpoints saved during distributed execution, in which the date of each (non-initial) local checkpoint c is such that $c.date = \max\{c'.date \mid c' \xrightarrow{zz} c\} + 1$. Let us associate with each local checkpoint c the global checkpoint $S = [c_1, ..., c_n]$ where c_i is the last local checkpoint of p_i, such that $c_i.date \leq c.date$. Then, S includes c and is consistent.*

As we can see, this theorem provides us with an operational tool (logical linear time) to design checkpointing algorithms. These algorithms are based on *implicit synchronization*. Independently from the other processes, each process can take local checkpoints (called *spontaneous* checkpoints) according to the needs of the upper layer applications. To prevent useless checkpoints, processes are required to add control information to application messages (let us notice that there are no pure control messages). This control information is then used by the receiver process to decide if this message reception has to entail the saving of a *non-spontaneous* (also called *forced*) local checkpoint. These algorithms are called *communication-induced checkpointing algorithms*. The interested reader will consult the following references where are presented such algorithms [31,32,33]. These algorithms differ in the notion of time used to capture causality and track zigzag paths. The most elaborate algorithms use vector time. A survey on checkpointing algorithms is presented in Chapter 12 of [62]. While some communication-induced checkpointing algorithms generate less forced checkpoints than others, it is shown in [9] that –from the point of view of the number of forced checkpoints– there is no optimal communication-induced checkpointing algorithm. Intuitively, this relies on the fact that when a process is forced to take a local checkpoint c, the communication pattern that will occur in the future will maybe make c not needed to obtain consistent global checkpoints, and this cannot be known in advance.

4 A Visit to Read/Write Registers

4.1 Atomic read/write register

Read/write registers are fundamental: they are the universal objects sequential computing is based on, namely, they are the cells of the Turing machine tape (that can only be written and read).

In a concurrency context the processes need to communicate, which means that the same register can be accessed concurrently by several processes. Hence different types of read/write registers have been defined by Lamport [41]. They differ in the values that a read operation returns in the presence of a concurrent writing. These are named safe, regular, and atomic. While regular (resp. atomic) registers offers an abstraction layer higher than safe (resp. regular) registers, they have the same computability power in asynchronous systems where any number of processes may crash (a crash is an unexpected and definite halt). Chapters 11-13 of [61] survey algorithms building multivalued multi-reader multi-writer

atomic registers from single-reader single-writer safe bits. The correctness of an atomic read/write register is defined by the following safety property.

– The operations invoked by the processes appear as if they have been executed sequentially (where a read returns the value of the closest preceding write), and this sequence respects real time order (i.e., if a read or write operation `op1` terminates before a read or write operation `op2` starts, then `op1` appears before `op2` in the sequence.

4.2 Read/write registers in crash-prone distributed systems

As read/write registers are the most basic objects of sequential computing, a first step of distributed computing consists in building them on top of an asynchronous crash-prone n-process system, where each pair of processes communicate through a reliable asynchronous channel.

On the negative side The first result is an impossibility result due to Attiya, Ben Or, and Dolev [5], who prove the following theorem (the proof is based on an *indistinguishability* argument [6]).

Theorem 3. *There is no algorithm that builds an atomic read/write register in an asynchronous message-passing system in which any number of processes may crash.*

On the positive side The authors of [5] have also shown in the same article that atomic read/write registers can be be built in asynchronous message-passing systems in which a majority of processes do not crash. Let us assume that local processing times have zero duration and message transfer delays are upper bounded by Δ (this bound remains always unknown to the processes, so it cannot be used in the algorithms). The upper bounds on operations are 2Δ (i.e., a round trip delay) for a write and 4Δ for a read. Moreover, in addition to read and written values, messages have to carry sequence numbers.

operation	concurrency/failure context	duration
write	always	2Δ
read	if no concurrent write	2Δ
read	if concurrent write	3Δ
read	if crashing concurrent write	4Δ

Table 1. Time efficiency of the algorithm described in [54]

Many other algorithms are described in the literature, that strive to reduce the size of the control information piggybacked on the messages that are exchanged, or the maximal duration of the read and write operations. Among

them, the algorithm described in [52] uses messages whose control information is reduced to two bits. The algorithm presented in [54] (joint work with A. Mostéfaoui and M. Roy) focuses on time efficiency. Its features are described in Table 1.

4.3 Read/write registers in the presence of Byzantine failures

While in the crash failure model, a process behaves correctly (i.e., executes correctly its algorithm), this is no longer the case when processes commit Byzantine failures [57]. In such a failure model, some process (maliciously or not) behave arbitrarily, i.e. they do have the behavior specified by their algorithm.

In this case, the multi-writer register model is inoperative as a Byzantine process can corrupt all registers. So, the appropriate model is then the single-writer multi-reader model. The memory is seen as an array $M[1..n]$, such that, while all processes can read all the registers, only p_i can write $M[i]$. Two important results are presented in [37]. The first one proves the following theorem.

Theorem 4. *There is no algorithm that builds an array of single-writer multi-reader atomic read/write registers in an n-process asynchronous message-passing system in which $n/3$ or more processes are Byzantine.*

The second result is a signature-free algorithm that builds an array $M[1..n]$ of single-writer multi-reader atomic read/write registers in an n-process asynchronous message-passing system in which less than $n/3$ processes are Byzantine.

Given such a Byzantine-tolerant memory, the solution to some problems becomes simple. As an example let us consider the write/snapshot problem. Processes can invoke once the operation write_snapshot(). This operation allows a process p_i to deposit a value and obtains a set of pairs $\langle j, u\rangle$ where u is a value deposited by p_j. The problem is defined by the following properties. Let $output_i$ be the set of pairs returned to p_i. A *non-faulty* process is a process that is not Byzantine.

- Termination. The invocation of write_snapshot(). by a non-faulty process terminates.
- Self-inclusion. If p_i is non-faulty and invokes write_snapshot(v), $\langle i, v\rangle \in output_i$.
- Containment. If p_i and p_j are non-faulty and both invoke write_snapshot(), then $output_i \subseteq output_j$ or $output_j \subseteq output_i$.
- Validity. If p_i and p_j are non-faulty and $\langle j, u\rangle \in output_i$, then p_j invoked the operation write_snapshot(u).

The reader can check that, assuming a Byzantine-tolerant array $M[1..n]$ of single-writer multi-reader atomic registers (hence assuming less than $n/3$ Byzantine processes) the algorithm described in Fig. 4 implements a write/snapshot object. Each process p_i manages two local arrays $aux1_i[1..n]$ and $aux2_i[1..n]$ in which it stores two consecutive asynchronous reading of the shared memory $M[1..n]$ (this is usually called *double-collect*).

```
operation write_snapshot(v_i) is                    (code for p_i)
(01)  M[i] ← v_i;
(02)  for x ∈ {1, ..., n} do aux1[x] ← M[x] end for;
(03)  for x ∈ {1, ..., n} do aux2[x] ← M[x] end for;
(04)  while (aux1 ≠ aux2) do
(05)      aux1 ← aux2;
(06)      for x ∈ {1, ..., n} do aux2[x] ← M[x] end for
(07)  end while;
(08)  output_i ← { ⟨j, aux1[j]⟩ | aux1[j] ≠ ⊥ };
(09)  return(output_i).
```

Fig. 4. Implementing a Byzantine-tolerant write/snapshot object [37]

5 A Fundamental Need: to Agree and Cooperate

5.1 Read/write registers are not universal in asynchronous crash-prone message passing systems

Looking for universality seems to be inherent to mankind. In informatics, programming languages, compilers and operating systems are well-known examples of this universality endeavor. As already indicated, from an object point of view, read/write registers are universal in sequential computing. Moreover, as we have seen, they can be built on top of crash-prone asynchronous message-passing systems in which a majority of processes do not crash. But are they universal in these crash-prone systems?

To this end let us consider all the objects that can be defined by a *sequential* specification (e.g., the set of traces on their operations that describe all their correct behaviors)[3]. Stacks and queues are such familiar objects. It appears that, while all these objects can be built in failure-free asynchronous message-passing systems, not all of them can be built in the presence of asynchrony and process crashes even if a majority of processes do not crash, i.e., even if read/write registers can be built. More generally, read/write registers are not universal in asynchronous message-passing systems as soon as even a single process can crash [23,36].

This is due to the fact that the processes need to agree on a single order in which the operations they invoke must be applied to the object. Such an agreement is captured by the so-called *consensus* object.

5.2 Consensus: a fundamental object

Consensus provides the processes with a one-shot operation denoted propose() which allows a process to propose a value v (input parameter) and obtains a value as the result of its invocation (we say a process decides a value). It is defined by the following properties in the crash failure model.

[3] The case of objects defined by a *concurrent* specification is addressed in [14]. A more practical approach is presented in [15].

– Termination. If a process invokes propose() and does not crash, it decides.
– Agreement. No two processes decide different values.
– Validity (non-triviality). The decided value is a proposed value.

Given a sequence of consensus objects, universal constructions can be designed for asynchronous read/write or message-passing systems in which a majority of processes do not crash (see e.g., [25,36,58,68] to cite a few). These constructions are based on the well-known *state machine replication* paradigm [40]. Each process proposes to successive consensus instances its current local view of the operations invoked but –from its point of view– not yet applied to the object. It follows from the properties of the consensus object that one of these views is decided and imposed to all the processes [64][4].

5.3 Bad news and good news

Bad news Despite its great practical interest to build distributed services encapsulated in a state machine (automaton), it appears that consensus can not be solved in the presence of asynchrony and the possible crash of even a single process. The first proof for message-passing systems was given in [23] (and it is usually named FLP according to thr names of its authors). The first proof for read/write registers was given in [43]. Many other proofs of this impossibility results have later appeared in the literature, e.g., [72].

Good news Several approaches have been proposed to *circumvent* the previous impossibility result. Among them there are the following ones.

– Restrict the asynchrony of the system [22].
– Enrich the system with information on failures. This is the failure detector-based approach introduced in [17]. Given an impossible problem to solve, it consists in providing the processes with the weakest information on crash failures that allows to solve it. As far as consensus is concerned, it has been shown in [16] that the *eventual leader* failure detector (denoted Ω) is the weakest failure detector that allow to solve consensus.
– Enrich the system with additional power provided with random numbers [7].
– Restrict the pattern of input vectors. This is the condition-based approach described below.

Several consensus algorithms based on each of the previous approaches are described in Chapter 17 of [64].

5.4 The condition-based approach

Intuition This approach was introduced in [49] (common work with S. Rajsbaum and A. Mostéfaoui). As a simple introductory example, let us consider the case of

[4] A generalization of the consensus object used to build a universal construction involving several objects is presented in [1]. This universal construction uses also k-set agreement objects [19].

binary consensus where only the values 0 and 1 can be proposed, and assume that (a) at most one process may crash, and (b) it is a priori known that one of the values is proposed by more than a majority of processes. In this case, consensus can be easily solved: each process sends its value to all others processes, waits until it has received values from $(n-1)$ processes and decides the majority value it has received. The additional knowledge on the 0/1 pattern of the input vector provides enough information to solve consensus. This is the essence of the condition-based approach: restrict the set of input vectors so that consensus can be solved, while having the the greatest possible set of input vectors.

Notations

- $\mathcal{V}$ denotes the set of values that can be proposed.
- $\mathsf{equal}(a, I)$ denotes the number of occurrences of the value a in the input vector I.
- $\mathsf{dist}(I1, I2)$ denotes the Hamming distance between the vectors $I1$ and $I2$ (the number of entries in which they differ).
- $\mathsf{max}(I)$ denotes the greatest value in the vector I.

x-Legality A set S of input vectors is x-legal if there is a function $h : S \mapsto \mathcal{V}$ with the following properties:

- $\forall\ I \in S : \#_{h(I)}(I) > x$,
- $\forall\ I1, I2 \in S : \big(h(I1) \neq h(I2)\big) \Rightarrow \big(\mathsf{dist}(I1, I2) > x\big)$.

Definition A condition C is a set of input vectors satisfying x-legality.

The intuition that underlies this definition is the following. Given a condition C, each of its input vectors I allows a proposed value to be selected in order to be the value decided by the processes. That value is extracted from an input vector by the function $h()$, namely $h(I)$ is the value decided from input vector I.

To this end, $h()$ and all vectors I of C have to satisfy some constraints. The first constraint states that the value that the processes have to decide from I (this value is $h(I)$) has to be present enough in vector I. "Enough" means "more than x times". This is captured by the first constraint defining x-legality: $\forall\ I \in C : \#_{h(I)}(I) > x$.

The second constraint states that, if different values are decided from different vectors $I1, I2 \in C$, then $I1$ and $I2$ must be "far apart enough" from one another. This is to prevent processes that would obtain different views of the input vector from deciding differently. This is captured by the second constraint defining x-legality: $\forall\ I1, I2 \in C : \big(h(I1) \neq h(I2)\big) \Rightarrow \big(\mathsf{dist}(I1, I2) > x\big)$.

An example of condition Let C^x_{max} be the set of input vectors defined as follows:

$$C^x_{\mathsf{max}} \stackrel{def}{=} \{I \mid \mathsf{equal}(a, I) > x \text{ where } a = \mathsf{max}(I)\}.$$

It is easy to see that the set of input vectors defining C^x_{max} satisfies the legality properties ($h(I)$ is $\mathsf{max}(I)$) and hence the set C_{max} is a condition. Moreover,

this condition is maximal, which means that, given any input vector $I' \notin C$, the set $C \cup \{I'\}$ is not a condition (it does not satisfy the legality properties. More developments on conditions and the associated computability/complexity hierarchy can be found in [49,50,64].

Condition-based consensus A condition-based consensus algorithm for asynchronous system in which processes communicate through read/write registers and up to $1 \leq t < n$ processes may crash is described in [49]. The condition must be t-legal. When $t < n/2$ this algorithm can easily be adapted to work in a message-passing system.

Condition-based approach vs error correcting codes It appears that conditions exhibit a strong connection relating consensus and error-correcting codes: each input vector I encodes a value, namely the value that has to be decided from I. In this sense an input vector can be seen as a code-word. More developments on this connection involving both crash failures and Byzantine failures can be found in [24] where, among other points, it is shown that %

- the impossibility of consensus in an n-process asynchronous system in which up to $f_c \geq 1$ process may crash and $f_b < n$ processes are Byzantine, and
- the impossibility to build (f_c, f_b)-perfect codes

are the "same" impossibility (n is the length of the code words, a crash corresponds to a missing digit –erasure–, and a value proposed by a Byzantine process corresponds to a modified digit).

5.5 The case of Byzantine processes

This section presents three recent results concerning Byzantine-tolerant consensus (common work with Z. Bouzid, A. Mostéfaoui, and H. Moumen).

Optimal Byzantine-tolerant binary consensus A round-based Byzantine-tolerant randomized binary Byzantine consensus algorithm, which closed a problem open since 30 years, is presented in [46]. More precisely, this algorithm has the following noteworthy features (no previous algorithm has all of them).Let t denote the maximal number of Byzantine processes.

- The algorithm requires $t < n/3$ and is consequently optimal with respect to t.
- The algorithm uses a constant number of communication steps per round.
- The expected number of rounds to decide is constant.
- The message complexity is $O(n^2)$ messages per round.
- Each message carries a type, a round number and a constant number of bits.
- Byzantine processes may re-order messages sent to correct processes.
- The algorithm uses a weak coin. *Weak* means here that, given an integer $d \geq 2$, (1) with probability $1/d$ the non-Byzantine processes obtain the value 0, (2) with probability $1/d$, the non-Byzantine obtain the value 1, and (3) with probability $(d-2)/d$, some non-Byzantine processes obtain the value 0 while the other non-Byzantine processes obtain the value 1. (A perfect common coin corresponds to the case $d = 2$.)

A pre-processing algorithm extending the above Byzantine-tolerant algorithm to multivalued consensus is presented in [53].

A weakest synchrony assumption for Byzantine-tolerant consensus As in case of crash failures, it is possible to enrich the system with synchrony assumption so that Byzantine-tolerant Consensus can be solved when $t < n/3$.

Each bi-directional communication channel is replaced by two uni-directional channels. (This is to be as general as possible as it becomes possible to associate different transfer delays with each direction of a bi-directional channel.)

Let us consider the channel connecting a process p to a process q. This channel is *eventually timely* if there is a finite time τ and a bound δ, such that any message sent by p to q at time $\tau' \geq \tau$ is received by q by time $\tau' + \delta$. Let us observe that neither τ nor δ are known by the processes.

An *eventual* $\langle t+1 \rangle$*bisource* is a non-Byzantine process p that has (a) eventually timely input channels from t correct processes and (b) eventually timely output channels to t correct processes (these input and output channels can connect p to different subsets of processes).

It is shown in [12] that the existence of an eventual $\langle t+1 \rangle$bisource is the weakest synchrony assumption that allows Byzantine-tolerant consensus to be solved. This article presents also an algorithm based on this assumption.

Never decide a value proposed only by Byzantine processes While the value decided in a any Byzantine-tolerant binary consensus algorithm is always a value proposed by a correct process [64], this is no longer the case for multivalued consensus. Nevertheless, some applications require to never decide a value proposed only by Byzantine processes. To answer this issue two approaches have been investigated.

- Accept to decide a default value (e.g., $\perp$) when not enough non-Byzantine processes propose the same value. This is the approach presented in [51] where is introduced the notion of *validated broadcast*.
- Always decide a value proposed by a correct processes. This is possible if and only if there is a value v that is proposed by at least $(t+1)$ non-Byzantine processes. A randomized algorithm based on this necessary requirement that always decide a value proposed by a correct process is presented in [47].

6 Miscellaneous: There is no End to Research

6.1 Symmetry breaking

Among the symmetry breaking problems [13], leader election is one of the most important. It was conjectured in a 1989 PODC article that three read/write registers were necessary to elect a leader in failure-free n-process system where communication is through atomic read/write registers, and the participation of each process is required. We refuted this conjecture in [27] where is presented an election algorithm which uses a single read/write register. Moreover the size of this register is bounded (i.e., there is no forever increasing values).

6.2 Anonymous memory

While the notion of anonymous processes has been studied since a long time, the notion of an anonymous memory has been introduced only very recently by G. Taubenfeld in [73].

Let us consider a shared memory REG made up of m atomic read/write registers. Such a memory can be seen as an array with m entries, namely $REG[1..m]$. In a non-anonymous memory system, for any index x, $1 \leq x \leq m$, if two or more processes invoke the address $REG[x]$ they access the very same register. This a priori agreement no longer exists in a memory-anonymous system. In such a system the very same address identifier $REG[x]$ invoked by a process p_i and invoked by a different processp_j does not necessarily refer to the same atomic register. More precisely, a memory-anonymous system with m registers is such that:

- For each process p_i, an adversary defines a permutation $f_i()$ over the set of indexes $\{1, 2, \cdots, , m\}$, such that when p_i addresses $REG[x]$, it actually accesses $REG[f_i(x)]$, and
- no process knows the permutations.

Despite the strong disagreement on memory addresses, anonymous memories allow us to model biological cell modification. It is shown in [59,60] how the process of genome-wide epigenetic modifications, which allows cells to utilize the DNA, can be modeled as an anonymous shared memory system where, in addition to the shared memory, also the processes (that are proteins modifiers) are anonymous. In such an anonymous memory context, we have recently solved basic concurrency-related problems. More precisely,

- deadlock-free mutual exclusion is solved in [2] where it is also shown that, assuming that the processes communicate by reading and writing anonymous registers, the predicate AM-Mutex "m is relatively prime with all the integers $\in \{2, \cdots, n\}$" (which relates the size m of the anonymous memory and the number n of processes) is a necessary and sufficient condition,
- leader election is solved [28], where it is also shown that leader election is possible if and only if $\mathsf{gcd}(m, n) = 1$.

6.3 Fully anonymous systems

In a fully anonymous system, (a) processes cannot be distinguished the ones from the others (they have no name, have the very same code, and the same initialization), and (b) all the registers of the anonymous memory are initialized to the same default value. So an important question that naturally comes to mind is: what can be done in such a concurrent computing model. We answered this question in [69] for mutual exclusion by

- showing that it is impossible on top of read/write registers only,
- designing a fully anonymous deadlock-free mutual exclusion algorithm based on read/write/compare&swap registers, and

– showed that the previous AM-Mutex predicate is a necessary and sufficient condition for such an algorithm in the full anonymity context.

Other algorithms solving consensus and more generally set agreement problems in fully anonymous systems are described in [70]. Computability issues in fully anonymous systems is addressed in chapter 5 of [66] and in [74].

6.4 Self-stabilization

Recently, thanks to E. Schiller, I became interested in an old topic introduced by Dijkstra in 1974 [20], namely self-stabilization [4,20]. We used this technique to make basic communication and agreement abstractions tolerant to transient failures (in addition to crash failures). A self-stabilizing set-constrained delivery broadcast algorithm is described in in [44] and an algorithm binary consensus algorithm is presented in [45].

7 By Way of Conclusion

Still in quest of concision and simplicity, I conclude with the last sentence of a letter from Blaise Pascal (French mathematician and philosopher, 1623-1662) to one of his friend "*Excuse me for having written such a long letter, I had not enough time to write a shorter one.*" [5]

Acknowledgments

A warm thank to all my coauthors and all other authors cited (or not) in this article. Without them, this article would not exist. A special thank to Armando Castañeda, Carole Delporte, Hugues Fauconnier, Antonio Fernández Anta, Eli Gafni, Vijay Garg, Vincent Gramoli, Rachid Guerraoui, Jean-Michel Hélary, Maurice Herlihy, Damien Imbs, Yoram Moses, Achour Mostéfaoui, Sergio Rajsbaum, Matthieu Roy, Elad Schiller, Julien Stainer, Gadi Taubenfeld, and Corentin Travers.

I also want to thank Bertrand Meyer for his kind invitation to write this article and Jean-Pierre Briot, Patrick Cousot and Marie-Claude Gaudel for a careful reading of a previous version of it.

[5] As Pascal was both a mathematician and philosopher, I can't resist the pleasure of quoting the following joke (but is it really a joke?):
- Question from A to B: Do you know the difference between a mathematician and a philosopher?
- After some time, answer of B: ... ???
- Answer from A to B: There is no difference, both use a sheet of paper and a pencil, ... [and after some time], oh, I was forgetting, the mathematician uses also an eraser!

Remark

As all of you, I have been asked the following question many times (often from PhD students): What is a good paper?

At the very beginning (when I was younger, i.e. in the previous millennium!) my answer was mainly based on an objective numerical criterion, namely, a good paper is "a paper with numerous citations". Later I was saying "a paper that won the best paper award in a top conference". Still later I was saying "a paper that won a prize devoted to more than ten years old papers", etc.

But over time, none of these integer-based definitions fully satisfied me, and I started thinking to the papers that I myself consider as very important papers ... and I discovered that those were papers I was a little bit *kindly jealous* ... not to be a co-author! This was because, those are papers I like to read (and reread) because they are nicely written, their content go beyond their technical content, they introduce new ideas in a simple and efficient way, and have a very strong impact on the community. This is the effect of good papers: everyone makes them "theirs", assimilating them and passing the essence of them to students.

References

1. Afek Y., Gafni E., Rajsbaum S., Raynal M., and Travers C., The k-simultaneous consensus problem. *Distributed Computing*, 22(3):185-195 (2010)
2. Aghazadeh Z., Imbs D., Raynal M., Taubenfeld G., and Woelfel Ph., Optimal memory-anonymous symmetric deadlock-free mutual exclusion. *Proc. 38th ACM Symposium on Principles of Distributed Computing (PODC'19)*, ACM press, pp. 157-166 (2019)
3. Aigner M. and Ziegler G., Proofs from THE BOOK (4th edition). Springer, 274 pages, ISBN 978-3-642-00856-6 (2010)
4. Altisen, K., Devismes, S., Dubois, S., and Petit, F., *Introduction to distributed self-stabilizing algorithms*, Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool Pub., 167 pages (2019)
5. Attiya H., Bar-Noy A. and Dolev D., Sharing memory robustly in message passing systems. *Journal of the ACM*, 42(1):121-132 (1995)
6. Attiya H. and Rajsbaum S., Indistinguishability. *Communications of the ACM*, 63(5):90-99 (2020)
7. Ben-Or M., Another advantage of free choice: completely asynchronous agreement protocols. *Proc. 2nd ACM Symposium on Principles of Distributed Computing (PODC'83)*, ACM Press, pp. 27-30 (1983)
8. Berlinski D., *The advent of the algorirhm.* Hartcourt Inc., 346 pages (2000)
9. Baldoni R., Hélary J.M., and Raynal M., Rollback-dependency trackability: a minimal characterization and its protocol. *Information and Computation*, 165(2):144-173 (2001)
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. Birman K.P. A response to Cheriton and Skeen's criticism of causal and totally ordered communication. *Operating Systems Review*, 28(1):11-21 (1994)

12. Bouzid Z., Mostéfaoui A., and Raynal M., Minimal synchrony for Byzantine consensus. *Proc. 34th ACM Symposium on Principles of Distributed Computing (PODC'15)*, ACM Press, pp. 461-470 (2015)
13. Castañeda A., Rajsbaum S., and Raynal M., Generalized symmetry breaking, tasks, and non-determinism in concurrent objects. *SIAM Journal of Computing*, 45(2):379-414 (2016)
14. Castañeda A., Rajsbaum S., and Raynal M., Unifying concurrent objects and distributed tasks: interval-linearizability. *Journal of the ACM*, 65(6), Article 45, 42 pages (2018)
15. Castañeda A., Rajsbaum S., and Raynal M. To appear in *Communications of the ACM*, 2022.
16. Chandra T.D., Hadzilacos V., and Toueg S., The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685-722 (1996)
17. Chandra T.D. and Toueg S., Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225-267 (1996)
18. Chandy K.M. and Lamport L., Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63-75 (1985)
19. Chaudhuri S., More choices allow more faults: set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132-158 (1993)
20. Dijkstra E.W., Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644 (1974)
21. Dijkstra E.W., Some beautiful arguments using mathematical induction. *Algorithmica*, 13(1):1-8 (1980)
22. Dolev D., Dwork C., and Stockmeyer L., On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77-97 (1987)
23. 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)
24. Friedman R., Mostéfaoui A., Rajsbaum S., and Raynal M., Distributed agreement problems and their connection with error-correcting codes. *IEEE Transactions on Computers*, 56(7):865-875 (2007)
25. Gafni E. and Guerraoui R., Generalizing universality. *Proc. 22nd Int'l Conference on Concurrency Theory (CONCUR'11)*, Springer LNCS 6901, pp. 17-27 (2011)
26. Gleick J., *The information: a history, a theory, a flood.* Harper Collin Pub., 529 pages (2011)
27. Godard E., Imbs D., Raynal M., and Taubenfeld G. Leader-based deanonymization of an anonymous read/write memory. *Theoretical Computer Science*, 836(10):110-123 (2020)
28. Godard E., Imbs D., Raynal M., and Taubenfeld G. From Bezout identity to space-optimal leader election in anonymous memory systems. *Proc. 39th ACM Symposium on Principles of Distributed Computing (PODC'20)*, ACM press, pp. 41-50 (2020)
29. Hammerman R. and Russell A.R., *Ada's legacy: cultures of computing from the Victorian to the digital age*. Morgan & Clapypool, 214 pages, ISBN 9781970001488 (2015)
30. Harel D. and Feldman Y., *Algorithmics: The spirit of computing (third edition)*, Springer, 572 pages, ISBN 978-3-642-27265-3 (2012)
31. Hélary J.M., Mostéfaoui A., Netzer R.H.B., and Raynal M., Communication-based prevention of useless checkpoints in distributed computations. *Distributed Computing*, 13(1):29-43 (2000)

32. Hélary J.-M., Mostéfaoui A., and Raynal M., Communication-induced determination of consistent snapshots. *IEEE Transactions on Parallel and Distributed Systems*, 10(9):865-877 (1999)
33. Hélary J.-M., Mostéfaoui A., and Raynal M., Interval consistency of asynchronous distributed computations. *Journal of Computer and System Sciences*, 64:329–349 (2002)
34. Hélary J.M., Netzer R.H.B., and Raynal M., Consistency issues in distributed checkpoints. *IEEE Transactions on Software Engineering*, 25(4):274-281 (1999)
35. Hélary J.M., Raynal M., Melideo M. and Baldoni R., Efficient causality-tracking timestamping. *IEEE Transactions on Knowledge and Data Engineering*, 15(5):1239-1250 (2003)
36. Herlihy M.P., Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124-149 (1991)
37. Imbs D., Rajsbaum S., Raynal M., and Stainer J. Read/Write shared memory abstraction on top of an asynchronous Byzantine message-passing system. *Journal of Parallel and Distributed Computing*, 93-94:1-9 (2016)
38. Knuth D.E., Ancient Babylonian algorithms. *Comm. of the ACM*, 15(7):671-677 (1972)
39. Kramer S. N., *History begins at Sumer: thirty-nine firsts in man's recorded history.* University of Pennsylvania Press, 416 pages, ISBN 978-0-8122-1276-1 (1956)
40. Lamport L., Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565 (1978)
41. Lamport L., On interprocess communication, Part I: basic formalism. *Distributed Computing*, 1(2):77-85, 1986.
42. Lamport L., Teaching concurrency, *ACM Sigact News*, 40(1):58-62 (2009)
43. Loui M. and Abu-Amara H., Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163-183, JAI Press Inc. (1987)
44. Lundström O., Raynal M., and Schiller E.M., Self-stabilizing set-constrained delivery broadcast. *40th IEEE Int'l Conference on Distributed Computing Systems (ICDCS'20)*, IEEE Press, 11 pages (2020)
45. Lundström O., Raynal M., and Schiller E.M., Self-stabilizing indulgent zero-degrading binary consensus. *Proc. 22th Int'l Conference on Distributed Computing and Networking (ICDCN'21)*, ACM Press, 10 pages (2021)
46. Mostéfaoui A., Moumen H., and Raynal M., Signature-free asynchronous binary Byzantine consensus with $t < n/3$, $O(n^2)$ messages, and $O(1)$ expected time. *Journal of ACM*, 62(4), Article 31, 21 pages (2015)
47. Mostéfaoui A., Moumen H., and Raynal M., Randomized k-set agreement in crash-prone and Byzantine asynchronous systems. *Theoretical Computer Science*, 709:80-97 (2018)
48. Mostéfaoui A., Perrin M., Raynal M., and Cao J., Crash-tolerant causal broadcast in $O(n)$ messages. *Information Processing Letters*, Vol. 151, 105837, 6 pages (2019)
49. Mostéfaoui A., Rajsbaum S., and Raynal M., Conditions on input vectors for consensus solvability in asynchronous distributed systems. *Journal of the ACM*, 50(6):922-954 (2003)
50. Mostéfaoui A., Rajsbaum S., Raynal M., and Roy M., Condition-based consensus solvability: a hierarchy of conditions and efficient protocols. *Distributed Computing*, 17(1):1-20 (2004)
51. Mostéfaoui A. and Raynal M., Intrusion-tolerant broadcast and agreement abstractions in the presence of Byzantine processes. *IEEE Transactions on Parallel and Distributed Systems*, 27(4):1085-1098 (2016)

52. 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)
53. Mostéfaoui A. and Raynal M., Signature-free asynchronous Byzantine systems: from multivalued to binary consensus with $t < n/3$, $O(n^2)$ messages, and constant time. *Acta Informatica*, 54:501-520 (2017)
54. Mostéfaoui A.,Raynal M., and Roy M., Time-efficient read/write register in crash-prone asynchronous message-passing systems. *Springer Computing*, 10(1):3-17 (2019).
55. Netzer R.H.B. and Xu J., Necessary and sufficient conditions for consistent global snapshots. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):165-169, (1995)
56. Neugebauer O., *The exact sciences in antiquity*. Brown University press, 240 pages (1957)
57. Pease M., Shostak R., and Lamport L. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234 (1980)
58. Rajsbaum S. and Raynal M., Mastering concurrent computing through sequential thinking: a half-century evolution. *Communications of the ACM*, Vol. 63(1):78-87 (2020)
59. Rashid S., Taubenfeld G., and Bar-Joseph Z., Genome wide epigenetic modifications as a shared memory consensus. *6th Workshop on Biological Distributed Algorithms (BDA'18)*, London (2018)
60. Rashid S., Taubenfeld G. and Bar-Joseph Z., The epigenetic consensus problem. *28th Int'l Colloquium on Structural Information and Communication Complexity (SIROCCO'21)*, Springer LNCS 12810, pp. 146–163 (2021)
61. Raynal M., *Concurrent programming: algorithms, principles and foundations*. Springer, 515 pages, ISBN 978-3-642-32026-2 (2013)
62. Raynal M., *Distributed algorithms for message-passing systems.* Springer, 515 pages, ISBN: 978-3-642-38122-5 (2013)
63. Raynal M., Réflexions désordonnées. *Bulletin 1024 de la Société Informatique de France*, volume 9:115-122 (2016) https://www.societe-informatique-de-france.fr/wp-content/uploads/2016/11/1024-no9-Raynal.pdf
64. Raynal M., *Fault-tolerant message-passing distributed systems: an algorithmic approach.* Springer, 550 pages, ISBN: 978-3-319-94140-0 (2018)
65. Raynal M., Distributed computability: a few results Masters students should know. *ACM Sigact News, Distributed Computing Column*, 52(2):92-110 (2021)
66. Raynal M., *Concurrent crash-prone shared memory systems: a few theoretical notions*. Morgan and Claypool, 113 pages, ISBN: 9781636393292 (2022)
67. Raynal M., Schiper A., and Toueg S., The causal ordering abstraction and a simple way to implement it. *Information Processing Letters*, 39(6):343-350 (1991)
68. Raynal M., Steiner J., and Tabenfeld G., Distributed universality. *Algorithmica*. 76(2):502-535, (2016)
69. Raynal M. and Tabenfeld G., Mutual exclusion in fully anonymous shared memory systems. *Information Processing Letters*, Vol. 158, 105938, 7 pages (2020) (Corrigendum to appear)
70. Raynal M. and Tabenfeld G., Fully anonymous consensus and set agreement algorithms. *Proc. 8th Int'l Conference on Networked Systems (NETYS'20)*, Springer LNCS 12129, pp. 314-328 (2020)
71. Reisig W., Informatics as science. *Enterprise Modelling and Information Systems Architectures*, 15(6):1-13 (2020)

72. Taubenfeld G., On the nonexistence of resilient consensus protocols. *Information Processing Letters*, 37(5):285-289 (1991)
73. Taubenfeld G., Coordination without prior agreement. *Proc. 36th ACM Symposium on Principles of Distributed Computing (PODC'17)*, ACM Press, pp. 325-334 (2017)
74. Taubenfeld G., Anonymous shared memory. *Journal of the ACM*, 69(4), Article 24, 30 pages (2022)