An Informal Visit to the Wonderful Land of Consensus Numbers and Beyond

Michel Raynal

Institut Universitaire de France
IRISA, Université de Rennes, 35042 Rennes, France
Department of Computing, Hong Kong Polytechnic University
raynal@irisa.fr

Abstract

Since its introduction by M. Herlihy in 1991, *consensus number* has become a central notion to capture and understand the agreement and synchronization power of objects in the presence of asynchrony and any number of process crashes. This notion has now become fundamental in shared memory systems, when one is interested in the design of universal constructions for high level objects defined by a sequential specification.

The aim of this survey is to be a guided tour in the wonderful land of consensus numbers. In addition to more ancient results, it also presents recent results related to the existence of an infinity of objects –of increasing synchronization/agreement power– at each level of the consensus hierarchy.

Keywords: Agreement, Asynchronous read/write system, Atomic operation, Concurrent object, Consensus, Consensus hierarchy, Crash failure, Deterministic object, Distributed computability, Progress condition, Sequential specification, k-Set agreement, Universal construction, Wait-freedom.

1 Introduction

Concurrent objects and asynchronous crash-prone read/write systems A concurrent object is an object that can be accessed (possibly simultaneously) by several processes. From both practical and theoretical point of views, a fundamental problem of concurrent programming consists in implementing high level concurrent objects, where "high level" means that the object provides the processes with a higher abstraction level than the atomic hardware-provided oper-

ations. While this notion of "high abstraction level" is well-known and well-mastered since a long time in the context of (sequential and parallel) failure-free systems [7], it is far from being trivial in failure-prone systems where it is still an important research domain.

This paper considers systems made up of n sequential asynchronous processes which, at the hardware level, communicate through memory locations (memory words also called registers) which can be accessed by atomic operations [32, 34], including the basic read and write operations. Moreover, it is assumed that, in any run, any number of processes may crash (a crash is an unexpected halting).

On progress conditions Deadlock-freedom and starvation-freedom are well-known progress conditions in failure-free asynchronous systems. As their implementation is based on lock mechanisms, they are not suited to asynchronous crash-prone systems. This is due to the fact that it is impossible to distinguish a crashed process from a slow process, and consequently a process that acquires a lock and crashes before releasing it can entail the blocking of the entire system.

Hence, new progress conditions for concurrent objects suited to crash-prone asynchronous systems have been proposed. The strongest progress condition, which is the one considered in this paper[1], is *wait-freedom* [22] (abbreviated WF in the following). Let O be the object that is built. This progress condition states that each invocation of an operation on O issued by a process that does not crash terminates, whatever the behavior of the other processes, which can be arbitrarily rapid, slow, or even crashed[2].

Universal object in failure-free systems Read/write registers are universal in sequential computing, which (according to Church-Turing thesis) means that everything that can be mechanically computed, can be computed from read/write registers (those are actually the cells of the tape of a Turing machine [50]). They are also universal in failure-free parallel systems. This comes from the fact that concurrent processes can cooperate thanks to *mutual exclusion* [16], which can be realized (in failure-free systems) on top of read/write registers [27, 40, 47].

Universal construction in the presence of asynchrony and process crashes The notion of a universal construction, for asynchronous crash-prone shared memory systems was introduced by M. Herlihy [22]. This notion addresses the construction of high level objects (a) defined from a sequential specification and (b)

[1] Other progress conditions, such as *non-blocking* [28] or *obstruction-freedom* [23] have been proposed for failure-prone systems. They are not considered in this article. The interested reader will consult appropriate textbooks, such as [27, 40, 47].

[2] This progress condition can be seen as an extension suited to failure-prone systems of the starvation-freedom progress condition defined for failure-free systems.

whose operations are total, i.e., any object operation returns a result (as an example, a push() operation on an empty stack returns the default value $\perp$).

A WF-*compliant universal construction* is an algorithm that, given the sequential specification of an object O (or a sequential implementation of it), provides a concurrent implementation of O satisfying the wait-freedom progress condition for all its operations, despite asynchrony and any number of process crashes (Fig. 1).

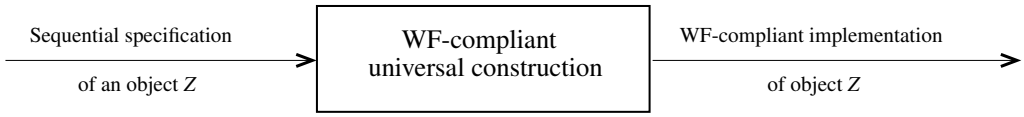

Figure 1: WF-compliant universal construction

It has been shown in [22, 33] that the design of a WF-compliant universal construction is impossible in asynchronous read/write systems where any number of processes may crash[3].

In failure-prone asynchronous (read/write or message-passing) distributed systems, the computability issues have a different nature than in failure-free asynchronous systems. As written in [25]: "*In sequential systems, computability is understood through the Church-Turing Thesis: anything that can be computed, can be computed by a Turing Machine. In distributed systems, where computations require coordination among multiple participants, computability questions have a different flavor. Here, too, there are many problems which are not computable, but these limits to computability reflect the difficulty of making decisions in the face of ambiguity, and have little to do with the inherent computational power of individual participants.*"

This means that asynchronous failure-prone systems need to be enriched with additional objects whose *computability power* is strictly stronger than the one of atomic read/write registers [41][4]. The objects that, together any number of read/write registers[5], allow to build a WF-compliant universal construction are said to be *universal*. As shown below the *consensus* object is universal.

[3]The first proof of such an impossibility was done in the context of asynchronous message-passing systems where even a single process may crash [18].

[4]Given a computing model (for example the finite state automaton model or the Turing machine model in sequential computing), the notion of *computability power* is on what can and what cannot be computed in this model. Differently, given a computing model, the notion of *computing power* refers to efficiency.

[5]It is show in [4] that any non-trivial object can implement atomic read/write registers in the wait-free task model.

Remark 1 As atomic read/write registers can be built on top of asynchronous message-passing n-process systems where up to $t < n/2$ processes may crash, the results presented in this article apply in these systems as soon as a majority of processes do not crash (see pages 75-169 of [42] for more details).

Remark 2 This article considers the classical shared memory distributed system model in which the concurrent objects to be implemented are deterministic. The case of non-deterministic objects is addressed in [38].

Remark 3 This article is no more than an informal introduction to the consensus number notion. The reader will find more precise developments of associated concepts and notions (such as *distributed task*, *long-lived task*, *computing model*, *oblivious object*, *object binding mode*, *robustness*, *deterministic vs. non-deterministic object*, etc.) in articles listed at the end of this article.

Content of the paper This paper is made up of five sections. Section 2 presents the consensus object and the associated consensus hierarchy notion (which allows us to capture the computability power of computing objects)[6]. Section 3 shows that there is an infinity of objects whose consensus number is 1, while their computability power is strictly increasing[7]. Section 4 shows that for any $x \geq 2$, there is an infinity of objects whose consensus number is x, while their computability power is strictly increasing[8]. Historically, the case $x \geq 2$ was investigated before the case $x = 1$ (respectively 2016 and 2018). The parlance "life beyond consensus" was introduced in [2]. Section 5 concludes the paper.

2 The Consensus Object and the Consensus Hierarchy

2.1 Consensus

Consensus object As already indicated, the notion of a *universal* object with respect to fault-tolerance was introduced by M. Herlihy [22]. An object type T is *universal* if it is possible to wait-free implement any object (defined by a sequential specification) in the asynchronous read/write model, where any number of processes may crash, enriched with any number of objects of type T. An algorithm providing such an implementation is called a *universal construction*. It is shown in [22] that *consensus* objects are universal. These objects, introduced

[6] The reference article is [22].
[7] The reference article is [14].
[8] The reference article is [2].

in [37], allow the processes to propose values and agree on one of them. More precisely, such an object provides the processes with a single operation, denoted propose(), that a process can invoke only once. This operation returns a value to the invoking process. When p_i invokes propose(v_i) we say that it "proposes the value v_i", and if v is the returned value we say that it "decides v". The consensus object is defined by the three following properties:

- Validity. The value decided by a process was proposed by a process.
- Agreement. No two processes decide different values.
- Termination. If a correct process invokes propose(), it decides a value.

Termination states that if a correct process invokes propose(), it decides a value whatever the behavior of the other processes (wait-freedom progress condition). Validity connects the output to the inputs, while Agreement states that the processes cannot decide differently. A sequence of consensus objects is used in the following way in a universal construction. According to its current view of the operations invoked on (and not yet applied to) the object O of type T that is built, each process proposes to the next consensus instance a sequence of operations to be applied to O, and the winning sequence is actually applied. A helping mechanism [8, 40] is used to ensure that all the operations on O (at least by the processes that do not crash) are eventually applied to O.

k-Set agreement A *k-set agreement* object (in short k-SA) is a simple an natural weakening of the consensus object [12]. It has the same Validity and Termination properties, but a weaker Agreement property, namely:

- Agreement. At most k different values are decided.

Hence, consensus is 1-set agreement. It is shown in [6, 26, 45] that it is impossible to implement k-set agreement on top of read/write registers, in the presence of asynchrony and any number of process crashes.

2.2 From consensus objects to a universal construction

Many algorithms have been proposed, which build a wait-free implementation of any object defined by a sequential specification (e.g. see [27, 40, 47]). This section presents a WF-compliant consensus-based universal construction inspired from the state machine replication paradigm (introduced in [31] in the context of failure-free systems), the process crash-tolerant total order broadcast algorithm presented in [9][9], and a helping mechanism implemented from atomic read/write registers. The reader will find a proof of it in [40]. As already said, and to make the presentation easier, it is assumed that the object O that is built is deterministic.

[9]Incidentally, the reader may notice that both the articles [9, 31] consider message-passing systems.

Sequential specification of the object The object O is assumed to be defined by a transition function $\delta()$. Let s be the current state of O and $\mathsf{op}(in)$ be the invocation of an operation $\mathsf{op}()$ on O, with input parameter in; $\delta(s, \mathsf{op}(in))$ outputs a pair $\langle s', r\rangle$ such that s' is the state of O after the execution of $\mathsf{op}(in)$ on s, and r is the result of $\mathsf{op}(in)$.

Local variables A process p_i manages locally a copy of the object, denoted $state_i$, an array $sn_i[1..n]$ where $sn_i[j]$ denotes the sequence number of the last operation on O issued by p_j locally applied to $state_i$. The local variables $done_i$, res_i, $prop_i$, k_i, and $list_i$, are auxiliary variables whose meaning is clear from the context; $list_i$ is a list of pairs of (operation, process identity); $|list_i|$ is its size, and $list_i[r]$ is its r-th element; hence, $list_i[r].op$ is an object operation and $list_i[r].proc$ the process that issued it.

```
when p_i invokes op(in) do
(1)   done_i ← false; BOARD[i] ← ⟨op(in), sn_i[i] + 1⟩;
(2)   wait (done_i); return(res_i).

Underlying local task T:    % background server task %
(3)   while (true) do
(4)       prop_i ← ϵ;   % empty list %
(5)       for j ∈ {1, ..., n} do
(6)           if (BOARD[j].sn > sn_i[j]) then
(7)               append (BOARD[j].op, j) to prop_i
(8)           end if
(9)       end for;
(10)      if (prop_i ≠ ϵ) then
(11)          k_i ← k_i + 1;
(12)          list_i ← CONS[k_i].propose(prop_i);
(13)          for r = 1 to |list_i| do
(14)              ⟨state_i, res_i⟩ ← δ(state_i, list_i[r].op);
(15)              let j = list_i[r].proc; sn_i[j] ← sn_i[j] + 1;
(16)              if (i = j) then done_i ← true end if
(17)          end for
(18)      end if
(19)  end while.
```

Figure 2: A wait-free consensus-based universal construction (code for process p_i)

Shared Objects The shared memory contains the following objects.

- An array $BOARD[1..n]$ of single-writer/multi-reader atomic registers. Each entry is a pair such that the pair $\langle BOARD[j].op, BOARD[j].sn\rangle$ contains

the last operation issued by p_j and its sequence number. Each read/write register $BOARD[j]$ is initialized to $\langle \perp, 0 \rangle$.

- An unbounded array $CONS[1..]$ of consensus objects.

Process behavior When a process p_i invokes an operation $\mathsf{op}(in)$ on O, it registers this operation together with its associated sequence number in $BOARD[i]$ (line 1). Then, it waits until the operation has been executed, and returns its result (line 2).

The array $BOARD$ constitutes the helping mechanism used by the background task of each process p_i. This task is made up two parts, which are repeated forever. First, p_i build a proposal $prop_i$, which includes the last operations (at most one per process) not yet applied to the object O, from its local point of view (lines 4-9 and predicate of line 6). Then, if the sequence $prop_i$ is not empty, p_i proposes it to the next consensus instance $CONS[k_i]$ line 12). The resulting value $list_i$ is a sequence of operations proposed by a process to this consensus instance. Process p_i then applies this sequence of operations to its local copy $state_i$ of O (line 14), and updates accordingly its local array sn_i (line 15). If the operation that was applied is its own operation, p_i sets the Boolean $done_i$ to true (line 16), which will terminate its current invocation (line 2).

Bounded wait-freedom versus unbounded wait-freedom Let us observe that this construction ensures that the operations issued by the processes are wait-free, but does not guarantee that they are *bounded* wait-free, namely, the number of steps (accesses to the shared memory) executed before an operation terminates is finite but not bounded. Consider a process p_i that issues an operation $\mathsf{op}()$, while $k1$ is the value of k_i. let and $k2 = k1+\alpha$ be such that $\mathsf{op}()$ is output by the consensus instance $CONS[k2]$. The task T of p_i must execute α times the lines 4-18 in order to catch up the consensus instance $CONS[k2]$ and return the result produced by $\mathsf{op}()$. It is easy to see that the quantity $(k2 - k1)$ is always finite but cannot be bounded.

A bounded construction is described in [22]. Instead of requiring each process to manage a local copy of the object, O is kept in shared memory and is represented by a list of cells including an operation, the resulting state, the result produced by this operation, and a consensus object whose value is a pointer to the next cell. The last cell defines the current value of the object.

2.3 The consensus hierarchy

Consensus numbers and consensus hierarchy The *consensus number* [22] associated with an object type T (denoted $\mathrm{CN}(T)$ in the following) is the greatest

positive integer n such that a consensus object can be built in an asynchronous crash-prone n-process system from any number of atomic read/write registers and any number of objects of type T. If there is no such finite n, the consensus number of T is $+\infty$. Hence, a type T such that $\text{CN}(T) \geq n$ is universal in a system of n (or less) processes.

It appears that the consensus numbers define an infinite hierarchy (also called "Herlihy's hierarchy") in which atomic read/write registers have consensus number 1, object types such as Test&Set, Fetch&Add, and Swap, have consensus number 2, etc., until object types such as Compare&Swap, Linked Load/Store Conditional (and a few others) that have consensus number $+\infty$. In between, read/write registers provided with m-assignment[10] with $m > 1$ have consensus number $(2m - 2)$.

Notations The following notations are used in the rest of the article.

- For $x \geq 1$, $\mathcal{CN}(x)$ denotes the set all the object types T whose such that $\text{CN}(T) = x$.
- If an object type has a single operation $\mathsf{op}()$, $\text{CN}(\mathsf{op})$ denotes its consensus number.
- If $T1$ and $T2$ are two object types such that $\text{CN}(T1) < \text{CN}(T2)$, we also write $T1 < T2$.
- If $CN(T) = x$ and O is an object of type T, we say that O is an x-consensus object (i.e., O allows consensus to be solved in an x-process system, but not in an $(x + 1)$-process system).
- Let T be an object type. $T < \mathcal{CN}(x)$ means that $\text{CN}(T) < x$, and similarly for $T > \mathcal{CN}(x)$.
- Let $A < B$ denote the fact that object A can be built in an n-process system where the processes communicate through read/write registers and objects B, while object B cannot be built from object A and read/write registers.

An object family covering the whole consensus hierarchy The object named *k-sliding read/write register* (in short RW_k) was introduced in [35] (a similar object was independently introduced in [17]). It is a natural generalization of an atomic read/write register, which corresponds to the case $k = 1$). Let *KREG* be such an object. It can be seen as a sequence of values, accessed by two atomic operations denoted *KREG*.$\mathsf{write}()$ and *KREG*.$\mathsf{read}()$.

[10] Such an assignment updates atomically m read/write registers. It is sometimes written $X_1, X_2, \cdots, X_m \leftarrow v_1, \cdots, v_m$ where the X_i are the registers, and each v_i the value assigned to X_i.

The invocation of $KREG$.write(v) by a process adds the value v at the end of the sequence $KREG$, while an invocation of $KREG$.read() returns the ordered sequence of the last k written values (if only $x < k$ values have been written, the default value $\perp$ replaces each of the $(k - x)$ missing values).

Hence, conceptually, an RW_k object is a sequence containing all the values that have been written (in their atomicity-defined writing order), and whose each read operation returns the k values that have been written just before it, according to the atomicity order. As already indicated, it is easy to see that, for $k = 1$, RW_k is a classical atomic read/write register. For $k = +\infty$, each read operation returns the whole sequence of values written so far. Let us notice that RW_∞ is nothing else than a ledger object [42].

It is shown in [35] that the consensus number of RW_k is k. Hence, from a computability point of view we have

$$\text{R/W registers} = \text{RW}_1 < \text{RW}_2 < \cdots < \text{RW}_k < \text{RW}_{k+1} < \cdots < \text{RW}_\infty.$$

2.4 A glance inside the consensus number land

Multiplicative power of consensus numbers The notion named *multiplicative power of consensus numbers* was introduced in [29]. It considers system models made up of n processes prone to up to t crashes, and where the processes communicate by accessing read/write atomic registers and x-consensus objects (with $x \leq t < n$). Let $ASM(n, t, x)$ denote such a system model. While the BG simulation [6] shows that the models $ASM(n, t, 1)$ and $ASM(t + 1, t, 1)$ are equivalent from a (colorless task) computability power point of view, the work presented in [29] focuses on the pair (t, x) of the system model parameters. Its main result is the following: the system models $ASM(n_1, t_1, x_1)$ and $ASM(n_2, t_2, x_2)$ have the same computability power if and only if $\lfloor \frac{t_1}{x_1} \rfloor = \lfloor \frac{t_2}{x_2} \rfloor$. This contribution, which complements and extends the BG simulation, shows that consensus numbers have a multiplicative power with respect to failures, namely the system models $ASM(n, t', x)$ and $ASM(n, t, 1)$ are equivalent (for colorless decision tasks) if and only if $(t \times x) \leq t' \leq (t \times x) + (x - 1)$.

Combining object types The consensus hierarchy considers that consensus must be built from read/write registers and objects of a given type T only. Hence the question "is it possible to *combine* objects with a *small* consensus number to obtain a new object with a greater consensus number?" As an example, let us consider thee two following object types $T1$ and $T2$, whose consensus number is 2 (see [17] for more developments).

- An object of type $T1$ can be read and accessed by the operation test&set(), which returns its current value and sets it to 1 if it contained 0.

- An object of type $T2$ can be read and accessed by the operation fetch&add2(), which returns the current value of the object, and increases it by 2.

Let us now consider an object type $T12$ which provides three operations: read(), test&set(), and fetch&add2(). The algorithm described in Fig. 3 (due to [17]) shows that a binary consensus object can be built from read/write registers and objects $T12$ in a crash-prone system of any number of processes. Binary consensus means that only the values 0 and 1 can be proposed[11]. We consequently have CN($T12$) = $+\infty$.

```
when p_i invokes propose(v) do
(1)   if (v = 0) then X.fetch&add2();
(2)                   if (X is odd) then return(1) else return(0) end if
(3)              else x ← X.test&set();
(4)                   if (x is odd) ∨ (x = 0) then return(1) else return(0) end if
(5)   end if.
```

Figure 3: A wait-free binary consensus algorithm from object type $T12$ (code for process p_i)

The internal representation of the binary consensus object is an object X of type $T12$, initialized to 0. According to the value it proposes (0 or 1), a process executes the statements of lines 2-3 or the statements of lines 4-5. The value returned by the consensus object is sealed by the first atomic operation that is executed. It is 0 if the first operation on X is X.fetch&add2(), and 1 if first operation on X is X.test&set(). The reader can check that, if the first operation on X is fetch&add2(), X becomes and remains even forever. If it is test&set(), X becomes and remains odd forever. In the first case, only 0 can be decided, while in the second case, only 1 can be decided.

Relaxing object operations In [46] the authors consider many classical objects (such as queues, stacks, sets) and relax the semantics of their operations in order to see if these relaxations modify the consensus number of the relaxed object, and consequently are more tolerant to the net effect of asynchrony and process failures.

As an example let us consider the well-known type Q (queue) defined the three following operations: enqueue(), which adds a value at the end of the queue, dequeue(), which returns the oldest value of the queue and suppresses it from the queue, and peek(), which returns the oldest value without modifying the content of the queue. The following relaxed queue type, denoted $Q_{a,b,c}$, was introduced

[11] This is not a problem as it is possible to build a multivalued consensus object from binary consensus objects, see [40].

and investigated in [46]. Each possible (statically defined) triple of the type parameters a, b, and c gives rise to an instance of a *relaxed* queue type, defined the three following atomic operations:

- $\mathsf{enqueue}_a(v)$ inserts the value v at any one of the a positions at the end of the queue[12].
- $\mathsf{dequeue}_b()$ returns and removes one of the values at the b positions at the end of the queue.
- $\mathsf{peek}_c()$ returns (without removing it) one of the values at the b positions at the end of the queue.

Whatever the operation, it returns a default value $\perp$ if the queue is empty. when the type parameter a, b, or c is equal to 0, the corresponding operation is not supported. When it is ∞ it means that the corresponding operation can add, remove/return a value at any position. It is easy to see that the object type $Q_{1,1,0}$ is the usual queue object (without $\mathsf{peek}()$ operation), whose consensus number is 2 [22]. Let us observe that the smaller the value of the parameter $a \geq 1$, $b \geq 1$, or $c \geq 1$, the stronger the constraint imposed by the corresponding operation. Among many others, the following results are shown in [46].

- The consensus number of $Q_{1,1,1}$ is ∞, while the consensus number of $Q_{\infty,1,1}$ is 2. This come from the fact that $\mathsf{enqueue}_\infty()$ allows a value to be inserted at any position, while $\mathsf{enqueue}_1()$ imposes a very constrained order on value insertions.
- The consensus number of $Q_{1,1,2}$ is 2 (this follows from the relaxed operation $\mathsf{peek}_2()$).
- For $a > 0$, the consensus number of $Q_{a,0,1}$ is $+\infty$.

The notion of power number of an object *Obstruction-freedom* is a progress condition progress condition (hence a termination property) introduced in [23]. It was later extended to k-obstruction-freedom in [48] as follows ($k = 1$ gives obstruction-freedom):

- Termination. If a set of at most k processes execute alone during a long enough time and do not crash, each of them terminates its operation.

Hence, k-obstruction-freedom states that, during long enough period during which the concurrency degree does not bypass k, the operations terminate. While wait-freedom is independent of both the concurrency pattern and the failure pattern, obstruction-freedom depend on them. More general *asymmetric* progress conditions have been introduced in [30]. The computational structure of progress conditions is investigated in [48].

[12] The position of an item (value) in a queue is the number of items that precede it plus 1.

The notion of the *power number* of an object type T (denoted PN(T)) was introduced in [48]. It is the largest integer k such that it is possible to implement a k-obstruction-free consensus object for *any* number of processes, using any number of atomic read/write registers, and any number of objects of type T (the registers and the objects of type T being wait-free). If there is no such largest integer k, PN(T) = $+\infty$.

Hence, the power number of an object type T establishes a strong relation linking k-obstruction-freedom and wait-freedom, when objects of type T are used. Let us remind that CN(T) is the consensus number of the objects of type T. It is shown in [48] that CN(T) = PN(T).

The notion of set agreement power As defined in [15], the set agreement power of an object type T is the infinite sequence $\langle n_1, ..., n_k, n_{k+1}, ...\rangle$, such that for any ≥ 1, n_k is the greatest number of processes for which it is possible to wait-free solve k-set agreement with any number of objects of type T and read/write registers. As an example, for $n \geq 2$, the set agreement power of the $(n-1)$-consensus object type is $\langle n_1, ..., n_k, n_{k+1}, ...\rangle$, where for all $k \geq 1$, $n_k = k(n-1)$ [13].

It is shown in [10] that at each level $\ell \geq 2$ of the consensus hierarchy, there are objects that, while they have the same set agreement power, are not equivalent (i.e., at least one of them cannot implement the other). This result has been extended to deterministic objects in [11].

From the process crash model to the crash-recovery model The consensus hierarchy in a crash-recovery model has first been addressed in [5]. This model assumes that a failure resets the local variables of a process to their initial values (the local variables include the program counter of the process), and preserves the state of the shared objects. It is shown in [5] that consensus remains sufficently powerfull to implement (in this model) any sequentially defined concurrent object.

The notion of *recoverable consensus* has been introduced in [21]. Such a consensus is defined by the classical Validity and Agreement properties of consensus and the following Termination property: Each time a process invokes a recoverable consensus instance, it returns a decision or crashes. This means that if a process invokes a recoverable consensus instance and, while executing it, crashes a finite number of times, it decides. It is shown in [21] that the consensus number of the Test&Set() operation (which is 2 in the crash failure model) is still 2 in the crash-recovery model if failures are simulataneous, but drops to 1 if failures are independent. As stated in [21], this captures the fact that, "when failures are simultaneous, a process recovers with more information regarding the states of other processes, than when failures are independent".

3 Life in the "Consensus Number 1" Land

This section presents an infinite family of deterministic objects, denoted WRN_3, WRN_4, ..., WRN_k, WRN_{k+1}, etc., such that

- none of them can be wait-free built from atomic read/write registers only,
- WRN_{k+1} can be wait-free built from WRN_k but cannot build it, and
- none of these objects can wait-free implement a 2-consensus in an n-process asynchronous crash-prone system.

It follows that this infinite countable family of objects are totally ordered by their computability power, are stronger than read/write registers (whose consensus number is 1), and are weaker than all the objects whose consensus number is greater or equal to 2. The results presented in this section are due to E. Daian, G.Losa, Y. Afek, and E. Gafni [14] and concern deterministic objects. The case of non-deterministic objects, for which there are similar results, was addressed in [38].

3.1 The family of "Write and Read Next" objects

The WRN object family (where WRN stands for *Write and Read Next*) is a generic family, in which each instance of the genericity parameter k ($k > 2$) gives rise to a specific object type denoted WRN_k.

A WRN_k object has a single atomic operation denoted $\mathsf{wrn}_k()$, which can be invoked at most once by a process. From an conceptual point of view, this object can be seen as an array $A[0..k-1]$ initialized to $[\bot, \cdots, \bot]$. A process p_i invokes $\mathsf{wrn}_k(i, v)$ where $i \in \{0, \cdots, k-1\}$ and v is a value to be stored in the WRN object. The effect of the invocation of $\mathsf{wrn}_k(i, v)$ is defined by the atomic execution of Algorithm 1, where it is assumed that $v \neq \bot$. The ring structure $\langle i, (i+1), ..., (k-1), 0, 1, ..., i\rangle$, and its use in the write of $A[i]$ followed by the read of $A[(i+1) \bmod k]$ is the key providing the computability power of a WRN_k object.

```
operation wrn_k(i, v) is   % i ∈ {1, ··· , k − 1}, v ≠ ⊥
(1)  A[i] ← v;
(2)  return(A[(i + 1) mod k]).
```

Algorithm 1: The operation $\mathsf{wrn}_k(i, v)$ (invoked by p_i)

It is easy to see that the object WRN_k is deterministic (namely, the value returned by $\mathsf{wrn}_k()$ and the new value of A depend on the previous value of A and the input parameters of the $\mathsf{wrn}_k()$ operation only).

3.2 Computability power of WRN_k in a k-process system

This section shows that a WRN_k object ($k > 2$) cannot be built from read/write registers (and is consequently stronger than them), and cannot solve consensus for two processes in a set of k processes. To this end it shows that, for any $k > 2$, it is possible to solve $(k, k-1)$-set consensus (i.e., $(k-1)$-set consensus in a set of k processes) from a WRN_k object, and WRN_k can be built from $(k, k-1)$-set consensus and atomic read/write registers. The result then follows from the fact that $(k-1)$-set consensus cannot be wait-free solved from read/write registers [6, 26, 45], and cannot solve consensus for two processes.

From a WRN_k object to $(k, k-1)$-set consensus Algorithm 2 realizes such a construction. It uses an underlying object WRN_k, accessed by k processes p_0, ..., p_{k-1} (where i is the index/identity of p_i). A process p_i first invokes $WRN_k.\mathsf{wrn}_k(i, v_i)$ where v_i is the value it proposes (line 1). Hence, it writes the entry i of the underlying WRN_k object and reads its next entry, namely $(i+1) \bmod k$ (Algorithm 1). Then (line 2), if the value it obtains from WRN_k is different from $\perp$, it returns it. Otherwise, it returns the value v_i it proposed.

```
operation propose(i, v_i) is   % code for p_i
(1)  aux ← WRN_k.wrn_k(i, v_i);
(2)  if (aux ≠ ⊥) then r ← aux else r ← v_i end if;
(3)  return(r).
```

Algorithm 2: The operation $\mathsf{propose}(i, v_i)$ of $(k, k-1)$-set agreement in a k-process system

Algorithm 2 is trivially wait-free. Let us also observe that, as the process indices are in $\{0, \cdots, (k-1)$ and no two processes have the same index, any entry of WRN_k can be written by a single process. Moreover, due to the content of WRN_k and line 2, it follows that only proposed values can be returned.

Let us consider any process p_j that decides. Such a process returns the value written by $p_{(j+1) \bmod k}$, or its own value v_j if $p_{(j+1) \bmod k}$ crashed before depositing its proposed value in WRN_k. As the invocations of $WRN_k.\mathsf{wrn}_k()$ are atomic (i.e., they appear as if they have been executed one after the other in a real time-compliant order), it follows that, the first process that invokes $WRN_k.\mathsf{wrn}_k()$ always returns its own value. Moreover, if all the processes decide, all the entries of WRN_k have been filled in, and the last process, say p_x, that executes $WRN_k.\mathsf{wrn}_k()$, returns the value written by $p_{(x+1) \bmod k}$. Hence, the value proposed by p_x is not decided, and consequently at most $(k-1)$ values are decided.

From $(k, k-1)$-set consensus to a WRN_k object This construction (not presented here, see [14]) starts from a solution to $(k, k-1)$-set consensus, which is first transformed into a $(k, k-1)$-strong set election object. This object is such that if a process p_i decides the value v_j proposed by a process p_j, then, if p_j decides, it decides also v_j (implementations are described in [6, 20]). The construction of a WRN_k object from a $(k, k-1)$-strong set election object uses additional snapshot objects [1], the consensus number of which is 1.

What has been shown The previous discussion has shown that, in an asynchronous k-process system, where any number of processes may crash, $(k, k-1)$-set agreement and WRN_k objects are computationally equivalent. Hence, as the computability power of $(k, k-1)$-set agreement is stronger than the one of read/write registers and is weaker than the one of objects whose consensus number is 2, the same follows from WRN_k objects in a k-process system.

3.3 When there are more than k processes

Where is the difficulty Let us now assume that there are $n > k$ processes, p_0, ..., p_{n-1}, and WRN_k objects, each being accessed by a specific set of k processes, e.g., p_{i_1}, ..., p_{i_k}. There are two cases according to the fact, for each WRN_k object, the subset of k processes that access it is statically or dynamically defined. We consider here the case where this set is statically defined. The reader interested in the dynamic case will consult [14].

Whatever the case, the important issue that has to be solved comes from the fact that the k entries 0, 1, ..., $(k-1)$ of the WRN_k object, do not necessarily correspond to the k indexes (belonging to the set $\{0, ..., n-1\}$) of the k that access the considered WRN_k object. This means that addressing issues must be solved to pair-wise associate the indexes of the k concerned processes with the k entries of a WRN_k object.

Index addressing in the static case Let $\mathsf{comb}(k, n)$ be the number of subsets of k elements taken from a set of $n > k$ elements. There are consequently $\mathsf{comb}(k, n)$ possible WRN_k objects, namely an object per subset of k different processes. Let us order all these subsets from 1 to $\mathsf{comb}(k, n)$, obtaining the subsets sbs_1, ..., $sbs_{\mathsf{comb}(k,n)}$. Moreover, let us order the process indexes in each subset sbs_x, according to their increasing values. Finally, for each $x \in \{1, ..., \mathsf{comb}(k, n)\}$, let $f_x(i)$, where i is a process index belonging to sbs_x, the position of i (starting from position 0) in the ordered subset sbs_x. Hence $f_x(i)$ is an index in $\{0, ..., k-1\}$, and for any two different indexes $i, j \in sbs_x$ we have $f_x(i) \neq f_x(j)$.

$(k, k-1)$-Set agreement in an n-process system A construction of a $(k, k-1)$-set agreement object in a system of n processes, is described in Algorithm 3. This construction is a simple "index reduction". Let sbs_x be the set of processes that invoke the considered $WRN_k(sbs_x)$ object, which is consequently denoted $WRN_k(sbs_x)$. The index mapping function $f_x()$ is known by the processes in sbs_x.

```
operation propose(i, v_i) is   % code for p_i, i ∈ sbs_x
(1)  i' ← f_x(i);
(2)  aux ← WRN_k(sbs_x).wrn(i', v_i);
(3)  if (aux ≠ ⊥) then r ← aux else r ← v_i end if;
(4)  return(r).
```

Algorithm 3: The operation propose(i, v_i) of $(k, k-1)$-set agreement in an n-process system

3.4 Infinite hierarchy inside the "Consensus Number 1" land

The object family $\{\text{WRN}_k\}_{k\geq 3}$ defines an infinite hierarchy As already said, it has been shown in [6, 26, 45][13] that it is not possible for n processes, $n \geq k \geq 2$, to build $(k, k-1)$-set agreement objects from atomic read/write registers. Moreover, as just seen, $(k, k-1)$-set agreement objects and WRN_k objects are equivalent (from a computability point of view) in an n-process system where $n \geq k \geq 3$. It follows that WRN_k objects cannot either be built from atomic read/write registers.

On another side, given n processes communicating through atomic read/write registers and $(k, k-1)$-set agreement objects where $k \geq 3$, it is not possible to solve consensus for two processes [13, 24, 30]. Hence it follows that it is not possible to solve consensus for two processes from WRN_k objects when $n \geq k \geq 3$, and consequently their consensus number is 1.

Finally, considering an n-process system, where $n \geq k+x$ and $x \geq 1$, $(k+x, k-1+x)$-set agreement objects can be built from $(k, k-1)$-set agreement objects and read/write registers, while $(k, k-1)$-set agreement objects cannot be built from $(k+x, k-1+x)$-set agreement objects [13, 24]. It follows from the previous observations that, in an n-process system where $n \geq k \geq 3$, WRN_{k+1} objects can be built from WRN_k objects, while WRN_k objects cannot be built from WRN_{k+1} objects.

Let us remind that $CN(2)$ denote any object whose consensus number is 2. The meaning of the symbol "$<$" was introduced in Section 2.3. Piecing together the previous observations we have:

$$\text{R/W Register} < \cdots < \text{WRN}_{k+1} < \text{WRN}_k < \cdots < \text{WRN}_3 < CN(2).$$

[13] These articles were foundational in introducing topology to capture the behavior of distributed computations.

The object WRN$_2$ Let p_0 and p_1 be two processes that access the object WRN$_2$. The value returned by process p_i, $i \in \{0,1\}$ when it invokes $\mathsf{wrn}(i, v_i)$ depends on the fact it is or not the first process to invoke it. According to the atomicity of WRN$_2$, if p_i is the first, its invocation $\mathsf{wrn}(i, v_i)$ returns the value it proposes, namely v_i, otherwise it returns the value previously deposited in WRN$_2$, by the other process. Hence, WRN$_2$ allows two processes to solve consensus, i.e., CN(WNR$_2$) = 2. From a consensus number hierarchy's point of view, we consequently have WRN$_3$ < WRN$_2$.

4 Life in Each "Consensus Number ≥ 2" Land

For each value of $m \geq 2$, this section presents a countable infinite family of objects, denoted AEG$_{m,2}$, AEG$_{m,3}$, ..., AEG$_{m,k}$, etc., such that, for $k \geq 2$, we have

- the consensus number of AEG$_{m,k}$ is m,
- AEG$_{m,k}$ can be wait-free implemented from AEG$_{m,k+1}$,
- AEG$_{m,k+1}$ cannot be wait-free implemented from AEG$_{m,k}$ objects and atomic read/write register in a system of = $mk + m + k$ processes.

It follows that, at each level $m \geq 2$ of the consensus hierarchy, there is an infinite countable family of objects that are totally ordered by their computability power. All the results presented in this section are due to Y. Afek, F. Ellen, and E. Gafni [2] (hence, the name "AEG" of these objects forged from the first letter of their surnames).

4.1 The family of AEG$_{m,k}$ objects

Let $m, k \geq 2$. The AEG$_{m,k}$ object seems partly inspired from the construction of k-set agreement objects in an n-process system from j-set agreement objects provided for free for any subset of m-processes. More precisely, an important result in this context is the following theorem due to [13, 24][14].

Theorem 1. *Let $n > k$ and $m > j$ be positive integers. It is possible to wait-free build k-set agreement objects in a system of n processes from j-set agreement objects accessed by m processes if and only if:*

$$(k \geq j) \wedge (n\ j \leq m\ k) \wedge (k \geq \mathsf{min}(j\lceil \tfrac{n}{m} \rceil, j\lfloor \tfrac{n}{m} \rfloor + n - m\lfloor \tfrac{n}{m} \rfloor)).$$

[14] This theorem was also instrumental in the design of an optimal k-set agreement algorithm in synchronous crash-prone message-passing systems [36], and in the establishment of a strong relation linking adaptive renaming and k-set agreement [20].

The AEG object family is a generic family, with two genericity parameter $n, k \geq 2$. Each value of m gives rise to a sub-family $\text{AEG}_{m,k}$, in which each instance of the parameter $k \geq 2$ give rise to a specific object.

An $\text{AEG}_{m,k}$ object has a single atomic operation denoted aeg_write(), which is invoked at most once by each process. From a conceptual point of view, this object can be seen as an array with k entries, namely $A[1..k]$, plus a counter. A process invokes $\text{aeg\_write}_{m,k}(v)$, where v is the value it wants to write in the $\text{AEG}_{m,k}$ object. The first $(mk + k - 1)$ invocations of $\text{aeg\_write}_{m,k}(v)$ return a value that has been written in A, while all the following invocations return the default value $\perp$.

More precisely, we have the following. Let us partition the sequence of the first $(mk + k - 1)$ invocations of $\text{aeg\_write}_{m,k}()$ into k sub-sequences of m invocations each, and a last sub-sequence of $(k - 1)$ invocations (see Fig. 4). Given the j-th invocation of $\text{aeg\_write}_{m,k}()$, Let CNT be an number of invocations $\text{aeg\_write}_{m,k}()$ previously executed (hence, $CNT = j - 1$).

- Considering the first sub-sequence of m invocations of aeg_write(), let a_1 be the input parameter of its first invocation. This value is written in $A[1]$. The other $(m - 1)$ invocations do not write. Moreover, all these m invocations of this first sub-sequence return a_1 Fig. 4).
- The same occurs for each sub-sequence of m invocations of aeg_write(), For the x-th sub-sequence, $2 \leq x \leq k$, let a_x be the input parameter of its first invocation. This value is written in $A[x]$. The remaining $(m - 1)$ invocations of this sub-sequence do not write, and all the m invocations of this x-th sub-sequence return a_x.
- Finally, For $mk + 1 \leq j \leq mk + k - 1$, the j-th invocation of aeg_write() does not write and returns the value in $A[mk + k - 1 - CNT]$, where CNT is the number of invocations of aeg_write() previously executed.

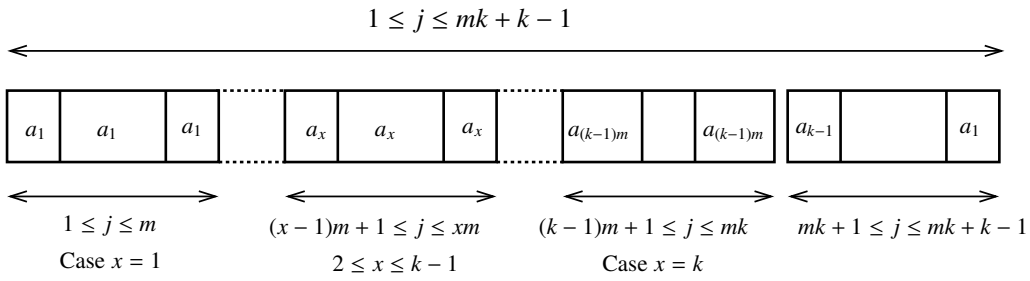

Figure 4: Value returned by the j-th invocation of $\text{aeg\_write}_{m,k}()$

Algorithm 4 is a simple translation of the previous description of $\text{aeg\_write}_{m,k}()$. Let us remind that this operation is atomic. It is easy to see that an $\text{AEG}_{m,k}$ object is deterministic.

```
operation aeg_write_{m,k}(v_i) is   % code for p_i
(1)  if (CNT = mk + k − 1) then return(⊥) end if;
(2)  if (CNT < mk)
(3)    then x ← ⌊CNT/m⌋ + 1;
(4)         if CNT = (x − 1)m then A[x] ← v end if
(5)    else  x ← km + k − (CNT + 1)
(6)  end if;
(7)  CNT ← CNT + 1;
(8)  return(A[x]).
```

Algorithm 4: The operation aeg_write$_{m,k}(v_i)$ invoked by p_i

4.2 The consensus number of an AEG$_{m,k}$ object is m

Assuming $m \geq 2$, let us consider the operation described in Algorithm 5, which uses an underlying AEG$_{m,k}$ object denoted $AEG_{m,k}$. It is easy to see that this algorithm solves consensus in an m-process system, and consequently the consensus number of AEG$_{m,k}$ is at least m.

```
operation propose_{m,k}(v_i) is   % code for p_i
(1)  r ← AEG_{m,k}.aeg_write(v_i);
(2)  return(r).
```

Algorithm 5: m-Process consensus on top of an AEG$_{m,k}$ object

In a very interesting way, replacing in Algorithm 5 the set of m processes by a larger set of $n = mk + k - 1$ processes, we obtain the more general theorem.

Theorem 2. *Let $n = mk + k - 1$ and $m, k \geq 2$. A k-set agreement object can be implemented from an* AEG$_{m,k}$ *object in an n-process system.*

While it is simple to show that the consensus number of the AEG$_{m,k}$ object is at least m, to show that it is exactly m is much more difficult, see [2] where is proved the following theorem.

Theorem 3. *Let $m, k \geq 2$. There is no deterministic algorithm implementing* binary consensus *from* AEG$_{m,k}$ *objects and read/write registers in an $(m + 1)$-process system.*

It follows from Algorithm 5 and Theorem 3 that the consensus number of AEG$_{m,k}$ is m.

Theorem 4. *Let $n \geq mk + k - 1$ and $m, k \geq 2$. An* AEG$_{m,k}$ *object cannot be implemented from m-*consensus *objects and read/write registers in an n-process system.*

This theorem can be easily proved by contradiction. Consider $n = mk + k - 1$, let us assume the contrary, namely, an $\text{AEG}_{m,k}$ object can be built from m-consensus objects in an n-process system. Using this $\text{AEG}_{m,k}$ object, It follows from Theorem 2 that a k-set agreement object can be built in an $(km+k-1)$-process system enriched with m-consensus objects. But, as $\frac{mk+k-1}{k} = m + 1 - \frac{1}{k} > \frac{1}{m}$, which contradicts Theorem 1.

4.3 An infinite hierarchy inside each "Consensus Number m" land, $m \geq 1$

$\text{AEG}_{m,k}$ can be implemented from $\text{AEG}_{m,k+1}$ Algorithm 6 presents a simple construction of an $\text{AEG}_{m,k}$ object from an $\text{AEG}_{m,k+1}$, from which it follows that (while they have the same consensus number, namely m) $\text{AEG}_{m,k+1}$ objects are at least as powerful as $\text{AEG}_{m,k}$ objects. This implementation is based on a specific initialization of the internal read/write registers implementing the underlying $\text{AEG}_{m,k+1}$ object. It is assumed that the value proposed by a process is a positive integer.

internal ad hoc initialization of the underlying $AEG_{m,k+1}$ object:
  $CNT \leftarrow m$; $A[1] \leftarrow 0$.

operation $\text{aeg\_write}_{m,k}(v)$ **is** % code for any p_i
(1) $aux \leftarrow AEG_{m,k+1}.\text{aeg\_write}_{m,k+1}(v+1)$;
(2) **if** $(aux > 0)$ **then** $r \leftarrow aux - 1$ **else** $r \leftarrow \perp$ **end if**;
(3) return(r).

Algorithm 6: $\text{AEG}_{m,k}$ object from $\text{AEG}_{m,k+1}$ object

This algorithm consists in a simple "elimination" of the first entry of the underlying array $A[1..k+1]$ implementing the $\text{AEG}_{m,k+1}$ object.

$\text{AEG}_{m,k+1}$ with respect to $\text{AEG}_{m,k}$ The following theorem is proved in [2], which states that an $\text{AEG}_{m,k+1}$ object is stronger than an $\text{AEG}_{m,k}$ object.

Theorem 5. *Let $m, k \geq 2$. An $\text{AEG}_{m,k+1}$ object cannot be implemented from $\text{AEG}_{m,k}$ objects and read/write registers in an $(mk + m + k)$-process system.*

An infinite hierarchy inside each "consensus number m" land, $m \geq 2$ It follows from the previous discussion that, at each level $m \geq 2$ of the consensus hierarchy, that, we have

$$\mathcal{CN}(m-1) < \text{AEG}_{m,2} \cdots < \text{AEG}_{m,k} < \cdots < \text{AEG}_{m,k+1} < \cdots < \mathcal{CN}(m+1).$$

5 Conclusion

The article constitutes a short visit to the notion of consensus number, which is a central notion as soon as one is interested in universal wait-free constructions of objects defined by a sequential specification. The reader interested in more developments can consult [41] for asynchronous crash-prone shared memory systems, and [43] for asynchronous crash-prone message-passing systems.

The following intriguing issue remains open: "is 1 a special number?" More precisely, the family of objects WRN_k was introduced to show there is life in the land of consensus number 1, while the family of objects $\text{AEG}_{m,k}$ was introduced to show there is life in each level $m \geq 2$ of the consensus hierarchy. The question is then "is there a single object family –instead of two– that show there is life at all the levels of the consensus hierarchy?"

Distributed universality is a fascinating topic. A more general notion of a k-universal construction was introduced in [19]. Such a construction considers the simultaneous construction of k objects (instead of only one), each defined by a specific type, and ensures that at least one of these objects progresses forever. This construction relies on k-SC objects (defined in [3]) instead of consensus objects. A still more general notion of (k, ℓ)-universal construction was proposed in [44] where $1 \leq \ell \leq k$, considers the case where, not at least one but at least ℓ objects progress forever, where ℓ is any predefined constant in $[1..k]$.

It follows from the results exposed in this introductory survey that, neither the notion of consensus number, nor the notion of set agreement power, characterizes the exact *computability power* of all the deterministic (and non-deterministic [38]) objects. On a close topic, the reader interested in the evolution of synchronization in the past fifty years can consult [39]. The interested reader will also find in [49] a study on the computability power of anonymous registers[15].

Acknowledgments

The author wants to thank the authors of all the papers cited in the reference list (and even a few others!). Without them this short introductory survey would not exist. A special thanks to J. Losa for a very interesting email exchange on topics related to this paper and more general ideas on fault-tolerant distributed computing models.

This work was partially supported by the French ANR project DESCARTES

[15] Among other results, it is shown in [49] that, while the consensus number of an anonymous read/write bit is 1, this object is computationally weaker than a non-anonymous bit and weaker than an anonymous read/write register, whose consensus numbers are also 1.

devoted to distributed software engineering (ANR-16-CE40-0023-03) and the Department of Computing of Hong Kong Polytechnic University.

References

[1] Afek Y., Attiya H., Dolev D., Gafni E., Merritt M., and Shavit N., Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873-890 (1993)

[2] Afek Y., Ellen F., and Gafni E., Deterministic objects: life beyond consensus. *Proc. 35th ACM Symposium on Principles of Distributed Computing (PODC'16)*, ACM Press, pp. 97-106 (2016)

[3] Afek Y., Gafni E., Rajsbaum S., Raynal M., and Travers C., The k-simultaneous consensus problem. *Distributed Computing*, 22(3):185-195 (2010)

[4] Bazzi R. A., Neiger G., and Peterson G. L., On the use of registers in achieving wait-free consensus. *Distributed Computing*, 10(3):117-127 (1997)

[5] Berryhill R., Golab W., and Tripunitara M., Robust shared objects for non-volatile main memory. *Proc. 19th Int'l Conference on Principles of Distributed Systems (OPODIS'15)*, LIPICs, Volume 46, pp. 20:1âĂŞ20:17 (2015)

[6] Borowsky E. and Gafni E., Generalized FLP impossibility results for t-resilient asynchronous computations. *Proc. 25th ACM Symposium on Theory of Distributed Computing (STOC'93)*, ACM Press, pp. 91-100 (1993)

[7] Brinch Hansen, P., *The origin of concurrent programming*. Springer, 534 pages, ISBN 0-387-95401-5 (2002)

[8] Censor-Hillel K., Petrank E., and Timnat S., Help! *Proc. 34th Symposium on Principles of Distributed Computing (PODC'15)*, ACM Press, pp. 241-250 (2015)

[9] Chandra T.D. and Toueg S., Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225-267 (1996)

[10] Chan D. Y. C., Hadzilacos V., and Toueg S., Life beyond set agreement. *Proc. 36th ACM Symposium on Principles of Distributed Computing (PODC'17)*, ACM Press, pp. 345-354 (2017)

[11] Chan D. Y. C., Hadzilacos V., and Toueg S., On the classification of deterministic objects via set agreement power. *Proc. 37th ACM Symposium on Principles of Distributed Computing (PODC'18)*, ACM Press, pp. 71-80 (2018)

[12] Chaudhuri S., More choices allow more faults: set consensus problems in totally asynchronous systems. *Information and Computation*, 105:132-158 (1993)

[13] Chaudhuri S. and Reiners P., Understanding the set consensus partial order using the Borowsky-Gafni simulation. *Proc. 10th Int'l Workshop on Distributed Algorithms*, Springer LNCS 1151, pp. 362-379 (1996)

[14] Daian E., Losa G., Afek Y., and Gafni E., A wealth of sub-consensus deterministic objects. *Proc. 32nd International Symposium on Distributed Computing (DISC'18)*, LIPICS 121, Article 17, 17 pages (2018)

[15] Delporte-Gallet C., Fauconnier H., Gafni E., and Kuznetsov P., Set consensus collections are decidable. *Proc. 20th Int'l Conference on Principles of Distributed Computing (OPODIS'16)*, LIPICS Vol. 70, Article 7, 17 pages (2016)

[16] Dijkstra E .W., Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569 (1965)

[17] Ellen F., Gelashvili G., Shavit N. and Zhu L., A complexity-based hierarchy for multiprocessor synchronization (Extended abstract). *Proc. 35th ACM Symposium on Principles of Distributed Computing (PODC'16)*, ACM Press, pp. 289-298 (2016)

[18] 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)

[19] Gafni E. and Guerraoui R., Generalizing universality. *Proc. 22nd Int'l Conference on Concurrency Theory (CONCUR'11)*, Springer LNCS 6901, pp. 17-27 (2011)

[20] Gafni E., Mostéfaoui, Raynal M., and Travers C., From adaptive renaming to set agreement. *Theoretical Computer Science*, 410:1328-1335 (2009)

[21] Golab W., The recoverable consensus hierarchy (Brief announcement). *Proc. 38th ACM Symposium on Principles of Distributed Computing (PODC'19)*, ACM Press, pp. 212-215 (2019). Full version: Recoverable consensus in shared memory, *ArXiv:1804.10597v2* (2018)

[22] Herlihy M. P., Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124- 149, (1991)

[23] Herlihy M.P., Luchangco V., and Moir M., Obstruction-free synchronization: double-ended queues as an example. *Proc. 23th Int'l IEEE Conference on Distributed Computing Systems (ICDCS'03)*, IEEE Press, pp. 522-529 (2003)

[24] Herlihy M.P. and Rajsbaum R., Algebraic spans. *Mathematical Structures in Computer Science*, 10(4):549-573 (2000)

[25] Herlihy M., Rajsbaum S., and Raynal M., Power and limits of distributed computing shared memory models. *Theoretical Computer Science*, 509:3-24 (2013)

[26] Herlihy M.P. and Shavit N., The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858-923 (1999)

[27] Herlihy M. and Shavit N., *The art of multiprocessor programming*. Morgan Kaufmann, 508 pages, ISBN 978-0-12-370591-4 (2008)

[28] Herlihy M.P. and Wing J.M., Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463-492 (1990)

[29] Imbs D. and Raynal M., The multiplicative power of consensus numbers. *Proc. 29th ACM Symposium on Principles of Distributed Computing (PODC'10)*, ACM Press, pp. 26-35 (2010)

[30] Imbs D., Raynal M., and Taubenfeld G., On asymmetric progress conditions. *Proc. 29th ACM Symposium on Principles of Distributed Computing (PODC'10)*, ACM Press, pp. 55-64 (2010)

[31] Lamport L., Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558-565 (1978)

[32] Lamport L., On interprocess communication, Part I: basic formalism. *Distributed Computing*, 1(2):77-85 (1986)

[33] Loui M. and Abu-Amara H., Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163-183, JAI Press (1987)

[34] Misra J., Axioms for memory access in asynchronous hardware systems. *ACM Transactions on Programming Languages and Systems*, 8(1):142-153 (1986)

[35] Mostéfaoui A., Perrin M., and Raynal M., A simple object that spans the whole consensus hierarchy. *Parallel Processing Letters*, Vol. 28(2), 9 pages (2018)

[36] Mostéfaoui A. Raynal M., and Travers C., Narrowing power vs efficiency in synchronous set agreement: relationship, algorithms and lower bound. *Theoretical Computer Science* 411:58-69 (2010

[37] Pease M., Shostak R., and Lamport L., Reaching agreement in the presence of faults. *Journal of the ACM*, 27:228-234 (1980)

[38] Rachman O., Anomalies in the wait-free hierachy. *Proc. 8th Int'l Workshop on on Distributed Algorithms (WDAG'94, Now DISC Symposium)*, Springer, LNCS 857, pp. 156-163 (1994)

[39] Rajsbaum S. and Raynal M., Mastering concurrent computing through sequential thinking: a half-century evolution. To appear in *Communications of the ACM* (2019)

[40] Raynal M., *Concurrent programming: algorithms, principles and foundations*. Springer, 515 pages, ISBN 978-3-642-32026-2 (2013)

[41] Raynal M., Distributed universal constructions: a guided tour. *Electronic Bulletin of EATCS (European Association of Theoretical Computer Science)*, 121:65-96 (2017)

[42] Raynal M., *Fault-tolerant message-passing distributed systems: an algorithmic approach. Springer*, 492 pages, ISBN: 978-3-319-94140-0 (2018)

[43] Raynal M., The Notion of universality in crash-prone asynchronous message-passing systems: a tutorial. *Proc. 38th Int'l Symposium on Reliable Distributed Systems (SRDS 2019)*, IEEE Press, 17 pages (2019)

[44] Raynal M., Stainer J., and Taubenfeld G., Distributed universality. *Algorithmica*, 76(2):502-535 (2016)

[45] Saks M. and Zaharoglou F., Wait-free k-set agreement is impossible: the topology of public knowledge. *SIAM Journal on Computing*, 29(5):1449-1483 (2000)

[46] Shavit N. and Taubenfeld G., The computability of relaxed data structures: queues and stacks as examples. *Distributed Computing*, 29(5):395-407 (2016)

[47] Taubenfeld G., *Synchronization algorithms and concurrent programming*. 423 pages, Pearson Education/Prentice Hall, ISBN 0-131-97259-6 (2006)

[48] Taubenfeld G., The computational structure of progress conditions. *Proc. 24th Int'l Symposium on Distributed Computing (DISC'10)*, Springer LNCS 6343, pp. 221-235 (2010)

[49] Taubenfeld G., The set agreement power is not a precise characterization for oblivious deterministic anonymous objects. *Proc. 26th Int'l Colloquium on Structural Information and Communication Complexity (SIROCCO'19)*, Springer LNCS 11639, pp. 290-304 (2019)

[50] Turing A. M., On computable numbers with an application to the Entscheidungsproblem. *Proc. of the London Mathematical Society*, 42:230-265 (1936)