A Look at Basics of Distributed Computing *

Michel Raynal

A Look at Basics of Distributed Computing [*]

Michel Raynal

Institut Universitaire de France
IRISA, Université de Rennes, 35042 Rennes Cedex, France
Department of Computing, HK Polytechnic University (PolyU), Hong Kong
`raynal@irisa.fr`

Abstract

This paper presents concepts and basics of distributed computing which are important (at least from the author's point of view), and should be known and mastered by Master students and engineers. Those include: (a) a characterization of distributed computing (which is too much often confused with parallel computing); (b) the notion of a synchronous system and its associated notions of a local algorithm and message adversaries; (c) the notion of an asynchronous shared memory system and its associated notions of universality and progress conditions; and (d) the notion of an asynchronous message-passing system with its associated broadcast and agreement abstractions, its impossibility results, and approaches to circumvent them. Hence, the paper can be seen as a guided tour to key elements that constitute basics of distributed computing.

Keywords: Asynchronous distributed computing, Consensus number, Locality of a computation, Process crash failure, Read/wite system, Synchronous communication, Wait-freedom.

---

[*]This paper is an invited "tutorial" paper presented at IEEE ICDCS 2016. (IEEE proceedings pages 2-11). It is dedicated to the memory of J.-Cl. Laprie (1944-2010).

1 By Way of an Introduction: Two Citations

For me, the first challenge for computing science is to discover how to maintain order in a finite, but very large, discrete universe that is intricately intertwined. And a second, but not less important challenge is how to mould what you have achieved in solving the first problem, into a teachable discipline: it does not suffice to hone your own intellect (that will join you in your grave), you must teach others how to hone theirs. The more you concentrate on these two challenges, the clearer you will see that they are only two sides of the same coin: teaching yourself is discovering what is teachable.

E. W. Dijkstra, *My hopes of computing science* (EWD 709) (1979)

Teaching is not an accumulation of facts.

L. Lamport, *Teaching concurrency*, ACM SIGACT News, 40(1):58-62 (2009)

2 What is the Essence of Distributed Computing

2.1 Distributed computing is about mastering uncertainty

Distributed computing[1] arises when one has to solve a problem in terms of 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 [56]. Hence, in one way or another, in any distributed computing problem, there are several computing entities, and each of them has to locally take a decision or compute a result, whose scope is global.

The geographical scattering of the computing entities, the (a)synchrony of their communication, their mobility, the fact that each entity initially knows only its own local inputs, the possibility of failures, etc., create *uncertainty* on the system state, in the sense that no computing entity can have an exact view of the current global state. This uncertainty, created by the *environment*, constitutes an *adversary* that the programmer cannot control but has to cope with.

Although distributed algorithms are often made up of a few lines, their behavior can be difficult to understand and their properties hard to state, prove, and implement. Hence, distributed computing is not only a fundamental topic of *Informatics*[2], but also a challenging topic where simplicity, elegance, and beauty are first-class citizens [20, 56].

2.2 The notion of a (distributed) task

The most fundamental notion of sequential computing is the notion of an *algorithm* implementing a mathematical function (left part of Figure 1). This gave rise to the notion of computability theory [68], and complexity theory [27], which are the foundations on which relies sequential computing.

Differently, the basic unit of distributed computing is the notion of a *task*, which was formalized in several papers (e.g, see [34, 35]). A task is made up of n processes $p_1, ..., p_n$ (computing entities), such that each process has its own input (let in_i denote the input of p_i) and must compute its own output (let out_i denote the output of p_i). Let $I = [in_1, \cdots, in_n]$ be an input vector (let us notice that a process knows only its local input, it does not know the whole input vector). Let $O = [out_1, \cdots, out_n]$ be an output vector (similarly, even if a process is required to cooperate with the other processes, it will compute only its local output out_i, and not the whole output vector). A task T is defined by a set $\mathcal{I}$ of input vectors, a set $\mathcal{O}$ of output vectors, and a mapping T from $\mathcal{I}$ to $\mathcal{O}$, such that, given any input vector

---

[1] Parts of this section are inspired from the position paper [58].

[2] As nicely stated by E.W. Dijkstra (1930-2002): "*Computer science is no more about computers than astronomy is about telescopes*". Hence, to prevent ambiguities, I use the word *informatics* in place of *computer science*. On a pleasant side, there is no more "computer science" than "washing machine science".
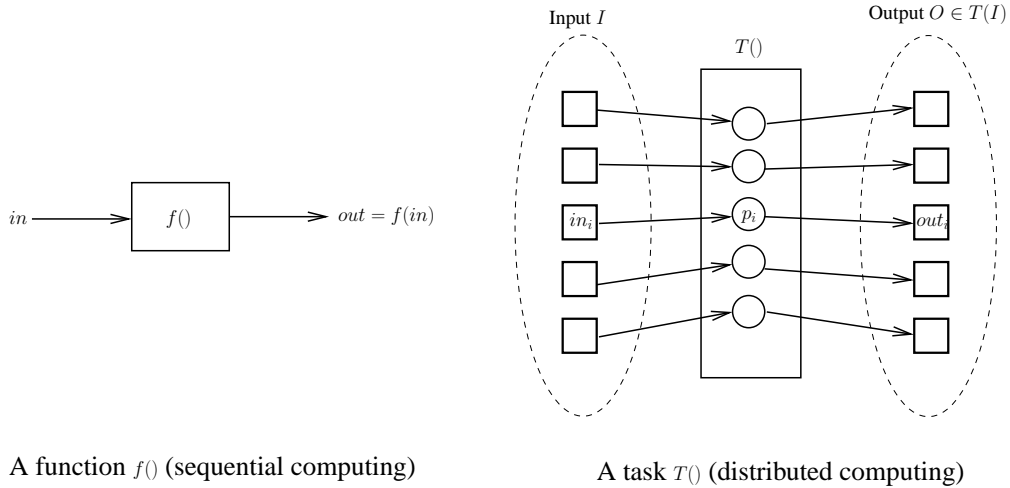
Figure 1: Function vs. task

$I \in \mathcal{I}$, the output vector O (cooperatively computed by processes) is such that $O \in T(I)$ (right part of Figure 1). The case $n = 1$ corresponds to sequential computing.

2.3 Distributed computing vs. parallel computing

This difference lies in the fact that a task is distributed by its very definition. This means that the processes, each with its own inputs, are geographically distributed and, due to this imposed distribution, need to communicate to compute their outputs. The geographical *distribution of the computing entities is a not a design choice*, it is an input of the problem which gives its name to *distributed computing*.

Differently, in parallel computing, the inputs are, by essence, centralized. When considering the left part of Figure 1, a function $f()$, and an input parameter x, parallel computing addresses concepts, methods, and strategies which allow to benefit from parallelism (multiple processing entities) when one has to implement $f(x)$ [7, 51]. The input x is given, and (if any) its initial scattering on distinct processors is not a priori imposed, but is a *design choice* aiming at obtaining efficient implementations of $f()$.

Any problem that can be solved by a parallel algorithm, could be solved (usually very inefficiently) by a sequential algorithm. Hence, the *essence* of parallel computing consists in mastering *efficiency*. Differently, the *essence* of distributed computing is not on looking for efficiency but on coordination in the presence of "adversaries" such as asynchrony, failures, locality, mobility, heterogeneity, limited bandwidth, etc.

Given a parallel application, it is of course possible that, due to a design choice, inputs are scattered on the processors by the application designer, and consequently distributed computing problems may appear at the implementation level of a parallel application.

2.4 The hardness of distributed computing

From a computability point of view, if the system is reliable, a distributed problem, abstracted as a task T, can be solved in a centralized way. Each process p_i sends its input in_i to a given predetermined process, which computes $T(I)$, and sends back to each process p_j its output out_j.

This is no longer possible if the presence of failures. Let us consider one of the less severe types of failures, namely process crash failures in an asynchronous system. One of the most fundamental impossibility result of distributed computing is the celebrated FLP result due to Fischer, Lynch, and Paterson [23]. This result states that it is impossible to design a deterministic algorithm solving the

3

basic *consensus* problem in an asynchronous distributed system in which even a single process may crash, be the underlying communication medium a message-passing network of a read/write shared memory (consensus is defined in Section 4.2). Roughly speaking, in a distributed setting, the local outputs depends on both the local inputs *and* the environment, while in a parallel setting, the outputs depends only on the inputs.

Hence, it appears that, in distributed computing, "*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*" [34]. As we can see, the essence of distributed computing is fundamentally different from the efficiency issues (task graph, scheduling, etc.) which motivated parallelism and constitutes its foundations [57].

3 Synchronous System, Locality, and Message Adversaries

3.1 Synchronous system model

Graph representation From a structural point of view, the system is represented by an undirected connected graph $G = (V, E)$, such $|V| = n$.

Each vertex represents a computing entity p_i, $1 \leq i \leq n$, which is a reliable sequential process (Turing machine enriched with the two communication operations send() and receive()). "Reliable" means that each process executes its local algorithm, without crashing or committing more severe failures (i.e., it never behaves arbitrarily).

Each edge of G represents a bidirectional reliable channel, on which the processes it connects can send or receive messages. "Reliable" means that there is neither loss, duplication, nor alteration of messages.

Synchrony assumption Synchrony is an abstraction that encapsulates and hides specific timing assumptions, so that algorithms can be written at a high abstraction level [45, 52, 54]. More precisely, assuming that the processes wake up simultaneously, each process p_i with its own input value in_i, they collectively execute a sequence of rounds, where a round r is made up of three sequential pases:

- A send phase during which each process sends a message to a subset of its neighbors.
- A receive phase during which each process receives the messages sent by its neighbors.
- A local computation phase during which a process modifies its local state according to the messages it received during the current round.

The fundamental synchrony property lies in the fact that a message sent during a round r is received during the very same round r. Hence, when a process progresses from round r to round $r + 1$, it knows that all the messages sent during round r are received, and all processes are progressing to round $r + 1$. Said differently, the processes advance in a lock-step manner, according to the round number, whose management is provided for free by the computation model. This model is sometimes called the LOCAL model.

3.2 Local algorithm

Let us assume that, at the first round, each process p_i sends to its neighbors the pair $\langle i, in_i \rangle$ at the first round, and then, at every round, it sends to its neighbors all the pairs it has learned during the previous rounds. As all processes start simultaneously, are reliable, and proceed synchronously, it follows that, after $x \geq 1$ rounds, each process p_i knows the pairs $\langle j, in_j \rangle$ of all the processes in its x-neighborhood (i.e., at distance at most x from it). Hence, after D rounds (where D is the diameter of the communication graph), p_i knows all the pairs $\langle , in_j \rangle$, and can consequently compute any function on the input vector $[in_1, \cdots, in_n]$.

Definition A distributed synchronous algorithm is *local* if its time complexity (measured as the number of rounds it has to execute in the worst case) is smaller than the graph diameter [43] (as an example a number of rounds polylogarithmic in the number or vertices, or even a constant). Hence, a fundamental issue of fault-free distributed synchronous computing consists in "classifying problems as locally computable [...] or not" [43].

An example As an example, let us consider the coloring of the vertices of a graph (i.e., any vertex must be assigned a color, such that any two neighbors have different colors and the number of colors is "small" or even minimal). A very elegant distributed synchronous algorithm, which colors the processes of a ring with at most three colors, is presented in [17]. This algorithm requires $\log^* n + 3$ rounds[3], which is asymptotically time optimal according to the $\Omega(\log^* n)$ lower bound established in [43].

The interested reader will find developments on locality in synchronous systems in many papers (e.g., [24, 39, 50]) and in the survey [66].

3.3 Message adversaries in synchronous systems

Historical perspective [4] *Message adversaries* have been introduced by Santoro and Widmayer in [63] to model and understand what they call *dynamic* transmission failures. The terms *ubiquitous* failures [64], *mobile* failure [46], and *transient* link failure [65], have also been used to capture similar network behaviors.

Aim The aim of this approach is to consider message losses as normal link behaviors (as long as messages are not corrupted). The notion of a message adversary is of a different nature than the notion of the fair link assumption. A fair link assumption is an assumption on each link taken separately, while the message adversary notion considers the network as a whole; its aim is not to build a reliable network but to *allow the statement of connectivity requirements* that must be met for a problem to be solved. Message adversaries allow us to consider topology changes not as anomalous network behaviors, but as an essential part of the deep nature of the system.

It follows that message adversaries allows to capture in a single concept both messages losses and the dynamicity (modification) of the communication network [13].

Definition A *message adversary* is a daemon which, at each round, can suppress messages (hence, these messages are never received). The adversary is not prevented from having a read access to the local states of the processes at the beginning of each round.

Let us associate a directed graph G^r with each round r, whose vertices are the processes, and there is an edge from p_i to p_j if the message sent at round r by p_i to p_j is not suppressed by the adversary. There is a priori no relation on the consecutive graphs G^r, G^{r+1}, etc. Among the possible daemon behavior, it can define G^{r+1} from the local states of the processes at the end of round r. Let $\mathcal{SMP}_n[adv : AD]$ denote the synchronous system whose communication is under the control of an adversary denoted AD. $\mathcal{SMP}_n[adv : \emptyset]$ denotes the synchronous system in which the adversary has no power (it can suppress no message), while $\mathcal{SMP}_n[adv : \infty]$ denotes the synchronous system in which the adversary can suppress all the messages at every round. It is easy to see that, from a message adversary and computability point of view, $\mathcal{SMP}_n[adv : \emptyset]$ is the most powerful synchronous system model, while $\mathcal{SMP}_n[adv : \infty]$ is the weakest. More generally, the more constrained the message adversary AD, the more powerful the synchronous system.

---

[3]Assuming $n > 1$, $\log_2^* n$ is the number of times the function "$\log_2$" must be applied in $\log_2(\log_2(\log_2 ...(\log_2 n)...))$ to obtain the value 1. Let us remember that $\log^*$(approx. number of atoms in the universe) $\simeq 5$.

[4]Parts of this section are from [59].

The spanning TREE message adversary Let TREE be the message adversary defined by the following constraint: at every round r, the communication graph G^r is an undirected spanning tree, i.e., the adversary cannot suppress the two messages –one in each direction– sent on the edges of G^r. Let $\mathcal{SMP}_n[adv : \text{TREE}]$ denote the corresponding synchronous system. As already indicated, for any r and $r' \neq r$, G^r and $G^{r'}$ are not required to be related, they can be composed of totally different sets of links.

Let us assume that each process p_i has an initial input v_i. It is shown in [38] that, $\mathcal{SMP}_n[adv : \text{TREE}]$ allows the processes to compute any computable function on their inputs, i.e., functions on the vector $[v_1, \ldots, v_n]$.

Solving this problem amounts to ensure that each input v_i attains each process p_j despite the fact the spanning tree can change arbitrarily from a round to the next one. This follows from the following observation. At any round r, the set of processes can be partitioned into two subsets: the set yes_i which contains the processes that have received v_i, and the set no_i which contains the processes that have not yet received v_i. As G^r is an undirected spanning tree (the tree is undirected because no message is suppressed on each of its edges), it follows that there is an edge of G^r that connects a process of the set yes_i to a process that belongs to the set no_i. So during round r, there at least one process of the set no_i which receives a copy of v_i, and will consequently belong to the set yes_i of the next round. It follows that at most $(n-1)$ rounds are necessary for v_i to attain all the processes.

The TOUR message adversary Let us assume that the underlying communication graph is complete (any pair of processes is connected by a channel). The message adversary denoted TOUR (for tournament) has been introduced in [1]. At any round, this adversary can suppress one message on each channel but not both: for any pair of processes (p_i, p_j), either the message from p_i to p_j, or the message from p_j to p_i, or none of them can be suppressed.

Let $A \simeq_T B$ mean that any task that can be computed in the model A, can be computed in the model B, and vice versa. Moreover, let $\mathcal{ARW}_{n,n-1}[fd : \emptyset]$ be the standard asynchronous wait-free read/write model (processes communicate by read/write registers only, and any number of processes may crash, see Section 4).

The following model equivalence is shown in [1]: $\mathcal{SMP}_n[adv : \text{TOUR}] \simeq_T \mathcal{ARW}_{n,n-1}[fd : \emptyset]$. This is an important result as it establishes a very strong relation linking message-passing synchronous systems where no process crashes but messages can be lost according to the adversary TOUR, with the basic asynchronous wait-free read/write model.

More relations linking asynchronous wait-free read/write systems, or asynchronous message-passing systems, both enriched with appropriate failure detectors [15], with synchronous systems weakened with message adversaries are described in [61].

4   Asynchronous Shared Memory System, and Universality

4.1   Asynchronous processes with register-based communication

Base t-resilient and wait-free read/write models These models are defined by a set of n asynchronous sequential processes, $p_1, ..., p_n$, which communicate through atomic read/write registers. The t-resilient model assumes that up to t processes may crash. We denote it $\mathcal{ASM}_{n,t}[\emptyset]$. The wait-free model is $\mathcal{ASM}_{n,n-1}[\emptyset]$ ($(n-1)$-resilient model). Hence, the wait-free model allows all processes, except one, to crash in an execution. Moreover, there is a strict hierarchy from $\mathcal{ASM}_{n,n-1}[\emptyset]$ (the weakest –and most– general model) to $\mathcal{ASM}_{n,0}[\emptyset]$ (the strongest –and reliable– shared memory model).

4.2 Universality in the wait-free model

The fundamental issue The fundamental issue of the wait-free model ($\mathcal{ASM}_{n,n-1}[\emptyset]$) lies in the following question [32]: *given atomic read/write registers and objects of some type T, is it possible to implement objects of some type T'?*

Universal object and universal construction Let $\mathcal{ASM}_{n,n-1}[T]$ denote $\mathcal{ASM}_{n,n-1}[\emptyset]$ enriched with objects of type T, and let $SeqSpec$ be the set of objects that can be defined by a sequential specification (e.g., stacks, queues, sets, graphs).

An object type T is $SeqSpec$-*universal* (in short universal) if any object of $SeqSpec$ can be built in $\mathcal{ASM}_{n,n-1}[T]$, (i.e., with atomic registers and objects of type T and despite asynchrony and any number of process crashes). An algorithm implementing such a generic construction is called a *universal construction* [32]. Examples of universal constructions can be found in textbooks such as [5, 55, 67].

Consensus object A consensus object provides the processes with a single operation, denoted propose(), that a process can invoke only once (hence it is a one-shot object).

This operation allows each process to propose a value and obtain (we say "decide") a value, such that the following properties are satisfied.

- Validity. If a process decides v, this decided value was proposed by a process.
- Agreement. No two processes decide different values.
- Integrity. A process decides at most once.
- Termination. If a process do not crash, it decides a value.

Validity relates the outputs to the inputs. Agreement states there is a single output. Termination states that at least the processes that do not crash must decide.

As cooperating processes have to agree in one way or another (otherwise the problem is only a control flow problem, as found in parallel computing), a lot of distributed computing problems rely on the consensus problem, or a variant of it.

Good news and bad news Two main results of distributed computing in the wait-free model are the following.

- The first result is that the consensus object is universal [32]. This means that any any object of $SeqSpec$ can be built in $\mathcal{ASM}_{n,n-1}[C]$, where C denote the consensus object type.
- The second result is the impossibility to implement consensus from atomic registers only, i.e., in $\mathcal{ASM}_{n,n-1}[\emptyset]$ [23, 32, 44]. Roughly speaking, this impossibility means that, contrarily to sequential computing, read/write registers are not powerful enough to solve some problems in the presence of asynchrony and failures.

Herlihy's hierarchy (consensus hierarchy) Fortunately, read/write registers are not the only type of objects that can be implemented in hardware. Numerous objects with atomic operations (such as test&set(), swap(), fetch&add(), compare&swap(), LL/SC(), etc.) are offered by multi-processors machines (multicore) to solve synchronization issues. (In the following we use the same name for an object type and its operation).

Let the *consensus number* of an object type T be the greatest integer n such that this object allows consensus to be implemented in $\mathcal{ASM}_{n,n-1}[T]$. if there is no such a greatest integer, the consensus number of T is $+\infty$.

The consensus number notion was introduced by Herlihy [32], who also introduced the following infinite hierarchy.

7

- The consensus number of atomic read/write registers is 1.
- The consensus number of the object types Test&Set, Fetch&Add, queue, stack, (and many others) is 2. Etc. There are objects with consensus number $n \in [3.. + \infty)$.
- The consensus number of the object types Compare&Swap, LL&SC, sticky bit (and others) is $+\infty$.

It follows from this hierarchy that, if one want to cope with process crashes when building a reliable application on top of a multicore architecture composed of n cores, one has has to consider a multicore providing an object type whose consensus number x is such that $x \geq n$.

Generalizing universality Basically, the previous notion of universality is on a single object. A more general notion of a k-universal construction has been introduced by in [26]. A k-*universal construction* is an algorithm that can be used to simultaneously implement k objects (instead of just one object), with the guarantee that at least one of the k constructed objects progresses forever. While Herlihy's universal construction relies on atomic registers and consensus objects, a k-universal construction relies on atomic registers and k-simultaneous consensus objects [2], which are equivalent to k-set agreement objects [16] in $\mathcal{ASM}_{n,n-1}[\emptyset]$. The k-set agreement is a weakening of consensus, which differs only in the Agreement property, namely, at most k different values can be decided by the processes ($k = 1$ for consensus).

An even more general notion of universality was introduced in [62], where is presented a k-universal construction which satisfies the following four desired properties:
- Among the k objects that are constructed, at least ℓ objects (and not just one) are guaranteed to progress forever.
- Any process that does not crash executes an infinite number of operations on each object that progresses forever (wait-freedom progress condition).
- The construction is contention-aware, which means that it uses only read/write registers in the absence of contention.
- The construction is generous with respect to the obstruction-freedom progress condition, which means that each process is able to complete any one of its pending operations on the k objects if all the other processes hold still long enough.

This construction is called a (k, ℓ)-universal construction. It uses a natural extension of k-simultaneous consensus objects, called (k, ℓ)-simultaneous consensus objects. These objects are shown to be universal (i.e., necessary and sufficient) for such a construction.

4.3 Weaker progress conditions, and abortable objects

Progress conditions weaker than wait-freedom The termination property stating that any invocation of an object operation issued by a process that does not crash, terminates, despite the behavior of the other processes (asynchrony and failures) is called *wait-freedom*. Unfortunately, in some cases, the design of a wait-free implementation of a concurrent object can be costly.

Hence, conditions weaker than wait-freedom have been proposed for the implementation of concurrent objects. They are the following ones.
- Non-blocking. If several processes invoke concurrently operations on an object, and at least one of these processes does not crash, at least one process returns from its invocation [36].
- Obstruction Freedom. If a process executes in isolation for a long enough period and does not crash, it returns from its operation invocation [33].

  "In isolation" means that, the other processes (which can be engaged in operations on the object) stop executing during the "long enough period". As the system is asynchronous, "long enough period" means a duration "during which a process can execute its operation".

let us observe that "non-blocking" is what is called "deadlock-freedom" in a failure-free context. Moreover, obstruction-freedom is a weaker property than non-blocking. Let us also notice that, as wait-freedom, non-blocking and obstruction-freedom implementations cannot use lock. This is due to the fact that, once an object is locked by a process p, it remains locked forever if p crashes before releasing the lock.

Example: obstruction-free $k$set agreement   An an example, let us consider the k-set agreement in the wait-free model $\mathcal{ASM}_{n,n-1}[\emptyset]$. As consensus, this problem is impossible to solve in this model when $k \leq n-1$. Hence, a way to solve it, is to weaken its termination property, requiring only that a process returns from its invocation if the obstruction-freedom property is satisfied (i.e., a process has enough time to execute its operation without being bothered by the other processes). An algorithm solving k-set agreement in an anonymous version of the wait-free model $\mathcal{ASM}_{n,n-1}[\emptyset]$ is presented in [9]. This algorithm uses $(n-k+1)$ multi-writer multi-reader atomic register, which is optimal.

Abortable objects   Another way to obtain more efficient implementations of an object is to relax the semantics of its operations as follows. The invocation of object operations executed in concurrency-free patterns must always terminate (if the invoking process does not crash). Otherwise they can abort (in which case, they do not modify the state of the object). Such objects are called *abortable* objects [11, 31, 55, 60].

It is also possible to combine abortable objects with the non-blocking progress property (see [55] for more details).

5   Asynchronous Message-passing System, Impossibility Results

5.1   Asynchronous system model

Definition   Let $\mathcal{AMP}_{n,t}[\emptyset]$ denote the n-process asynchronous message-passing computation model. In such a model, each process is sequential and asynchronous and up to t processes may crash in an execution. Moreover, each pair of processes is connected by an asynchronous reliable bi-directional channel. "Reliable"means that no message can be lost, duplicated, created from thin air, or modified. "Asynchronous" means that message transfer delays are arbitrary, may vary with time, but are finite. This is the classic model used in textbooks (e.g., [5, 10, 45, 53, 56]).

Reliable broadcast in $\mathcal{AMP}_{n,t}[\emptyset]$ One of the very most basic problems encountered in $\mathcal{AMP}_{n,t}[\emptyset]$ consists in implementing a reliable broadcast communication abstraction. Namely, to broadcast a message m a process p needs to sent m to all the processes. If p crashes during these sendings, only a subset of the non-crashed processes receive the message, and consequently the broadcast is unreliable.

A reliable broadcast ensures that all the correct processes (i.e., the processes that do not crash) deliver the same set of messages S, and this set includes –at least– all the messages they broadcast. This means that, if a process crashes, each of its messages is delivered by all or none of the correct processes. Moreover, a process that crashes deliver a subset of S (the set messages delivered by the correct processes). This communication abstraction was cleanly defined in [30]. The reader will find in the monograph [53] distributed algorithms implementing this abstraction in various message-passing models prone to both process crashes and channel failures.

From message-passing to read/write registers   Another of the most basic problems of asynchronous message-passing systems consists in building an atomic read/write register.

It is shown in [4] that $t < n/2$ is a necessary and sufficient condition to simulate a read/write register in $\mathcal{AMP}_{n,t}[\emptyset]$, hence (from a notation point of view) such a register can be implemented only in the restricted message-passing model $\mathcal{AMP}_{n,t}[t < n/2]$.

An algorithm (named ABD in the literature, according to the names of its authors) that implement a register in $\mathcal{AMP}_{n,t}[t < n/2]$ is described in [4]. This algorithm is based on majority quorums. The fact that these quorums include a majority of correct processes, associated with the rule "a reader has to write the value it returns" allows read operations to always obtain correct values. If each message is assumed to take Δ time units, a write operation require 2Δ time units, and a read operation requires 4Δ time units. A very recent algorithm, described in [49], improves the time duration of the read operation, which requires 2Δ time units in "good circumstances" and up to 4Δ time units only in "bad" circumstances.

Universality in $\mathcal{AMP}_{n,t}[\emptyset]$ The notion of universality introduced in Section 4.2 translates as follows in $\mathcal{AMP}_{n,t}[t < n/2]$ (the requirement $n < n/2$ is due to the fact that registers appear in the definition of the universality notion): *How to duplicate a state machine*?

This problem was posed first by Lamport in the context of fault-free message-passing systems [41]. In the context of $\mathcal{AMP}_{n,t}[t < n/2]$, it amounts to build a reliable broadcast abstraction such that all processes receive in the same order all the operations on the object that is built. In this way, they can apply the same sequence of operations to their local copies, which ensures their mutual consistency. Such a broadcast abstraction is called *Total Order Reliable* (TO-reliable) broadcast.

It appears that the construction of TO-reliable broadcast relies on consensus, namely, the processes have to agree on the order in which operations must be applied to their local copies, and this is a typical agreement problem. As consensus is impossible to solve in $\mathcal{AMP}_{n,1}[\emptyset]$ (i.e., even a single process may crash) [23]), TO-reliable broadcast is impossible to implement in $\mathcal{AMP}_{n,t}[t > 0]$.

5.2 A fundamental dilemma: symmetry breaking

When considering the basic distributed computing $\mathcal{AMP}_{n,t}[\emptyset]$, A fundamental dilemma is the following.

- On the one side, *all processes are equal* in the sense that any subset including up to t of them may crash, and it is not known in advance which processes will crash in a given run.

- On the other side, solving some problem requires that, *during some period of time, some process(es) be "more equal"* than the others. Such processes are usually called *leaders*.

This means that *symmetry breaking in the presence asynchrony and failures* is a fundamental issue in distributed computing. (This was known in some societies as "Primus inter pares".)

5.3 Circumventing impossibility results

Possible approaches Several approaches have been proposed to address impossibility results in the model $\mathcal{AMP}_{n,t}[t > 0]$. We focus here only on the consensus problem in $\mathcal{AMP}_{n,t}[t < n/2]$. Four main approaches have been investigated.

- Enrich the system with randomization, and weaken accordingly the termination property [6]. This allows the non-determinism created by the environment (asynchrony and failures) to be solved [57].

- Restrict the asynchrony of the system [21, 22].

- Restrict the space of the input vectors which can be proposed by the processes [48]. This approach established a strong connection relating error-correcting codes and agreement problems [25].

10

- Introduce failure detectors, i.e., devices that provide each process with information on failures. According to the problem to be solved and the quality of this information, several classes of failures detectors can be designed [15]. Actually, failure detectors can be seen as objects that abstract underlying synchrony assumptions.

The weakest failure detector to solve consensus   This failure detector was introduced in [14], where it is called Ω. "Weakest means" that given any information on process failures, which allows consensus to be solved, it is possible to build Ω.

This failure detector Ω provides each process p_i with a read-only local variable $leader_i$, which always contains a process identity, and satisfies the following "eventual leadership" property: There is a time τ, after which the variables $leader_i$ of all the processes that do not crash contain forever the same value id, and this identity is the identity of a correct process.

It is important to notice that, during an arbitrary long, but finite, period, each local variable $leader_i$ can behave arbitrarily. Moreover, the time instant τ is never explicitly known by the processes. Algorithms implementing Ω in $\mathcal{AMP}_{n,t}[\emptyset]$ with various underlying synchrony assumptions are presented in [53]. Ω can be seen as a formal definition of the leader service used in Paxos [42].

The correct information on failures provided by Ω is eventual, and consists in a single identity, which is not the one of a faulty process. No information on the state (correct or faulty) of other processes is needed to solve consensus. As we can see, Ω solves the dilemma stated previously.

Indulgent algorithms   The class of *eventual* failure detectors (hence Ω) has the following noteworthy feature. An algorithm based on such a failure detector always terminates (and produces a correct result), if the failure detector it relies on behaves as described by its specification. If the implementation of the failure detector never satisfies its specification, the algorithm may not terminate, but if it terminates, it always produces a correct result. Such algorithms are said to be *indulgent* with respect to their failure detectors [28, 29].

5.4   Process adversaries in asynchronous systems

The notion of a *process adversary* originated in [37], and was later generalized and formalized in [19]. A computability-oriented survey can be found in [40]. This notion generalizes the notion of a t-resilient algorithm.[5]

Definition   A *process adversary* A is a set of sets of processes. Given such a set A, and a problem P, the aim is to design an algorithm that (a) never violates the safety property of P, and (b) terminates in all the executions in which the set of non-faulty processes is a set in A.

As an example, Let us considers a system with four processes denoted p_1, ..., p_4. The set $A = \{\{p_1, p_2\}, \{p_1, p_4\}, \{p_1, p_3, p_4\}\}$ defines an adversary. An algorithm A-resiliently solves a problem if it terminates in all the executions where the set of non-faulty processes is exactly either $\{p_1, p_2\}$ or $\{p_1, p_4\}$ or $\{p_1, p_3, p_4\}$. This means that an A-resilient algorithm is not required to terminate in an execution in which the set of non-faulty processes is exactly the set $\{p_3, p_4\}$ or the set $\{p_1, p_2, p_3\}$.

The main interest of this notion is the fact that it allows the express termination conditions where process crashes are not uniform. It modifies the computation model, in the sense that not all processes are seen as equal, and do not fail in an independent way.

---

[5]This section borrows parts from [34].

An example: core and survivors sets A *core C* is a minimal set of processes such that, in any execution, some process in C does not fail. A *survivor* set S is a minimal set of processes such that there is a execution in which the set of non-faulty processes is exactly S. Let us observe that cores and survivor sets are dual notions (any of them can be obtained from the other one).

As an example let us consider a system of 4 processes p_1, p_2, p_3 and p_4 where two cores are defined, namely $\{p_1, p_2\}$ and $\{p_3, p_4\}$. The corresponding survivor sets are $\{p_1, p_3\}$, $\{p_1, p_4\}$, $\{p_2, p_3\}$ and $\{p_2, p_4\}$. (Borrowing the quorum terminology and considering cores as *quorums*, the corresponding survivor sets are their *anti-quorums*). Cores and survivor sets are assumed to be initially known by alll the processes.

6 Conclusion

The aim of this invited talk was to present some ideas and concepts which belongs to the basics of distributed computing. Due to time and page limitations, important ideas and concepts have not been addressed here (where the term "important" must be understood as subjective and engaging only the author).

Much more concepts belong to the basics of distributed computing, and still deserve work to be fully understood and mastered. I cite only three of them: anonymous/homonymous systems (e.g., [3, 8]); Byzantine failures in agreement problems (e.g., [47]); and dynamic systems (e.g., [12]).

Acknowledgments

References

[1] Afek Y. and Gafni E., A simple characterization of asynchronous computations. *Theoretical Computer Science*, 561: 88-95 (2015)

[2] Afek Y., Gafni E., Rajsbaum S. Raynal M., and Travers C. The k-simultaneous consensus problem. *Distributed Computing*, 22(3):185-196, 2010

[3] Arevalo S., Fernandez A., Imbs D., Jimenez E., and Raynal M., Failure detectors in homonymous distributed systems (with an application to consensus). *Journal of Parallel and Distributed Systems*. 83:83-95 (2015)

[4] Attiya H., Bar-Noy A. and Dolev D., Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):121-132 (1995)

[5] Attiya H. and Welch J.L., *Distributed computing: fundamentals, simulations and advanced topics, (2nd Edition)*, p. 414. Wiley-Interscience (2004) ISBN 0-471-45324-2

[6] 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)

[7] Blazewicz J., Pesch E., Trystram D., and Zhang G., New perspectives in scheduling theory. *Journal of Scheduling*, 18(4):333-334 (2015)

[8] Bonnet F. and Raynal M., Anonymous asynchronous systems: the case of failure detectors. *Distributed Computing*, 26(3):141-158 (2013)

[9] Bouzid Z., Raynal M., and Sutra P., Anonymous obstruction-free (n, k)-set agreement with $(n-k+1)$ atomic read/write registers. *Proc. 19th Int'l Conference On Principles Of Distributed Systems (OPODIS'15)*, 17 pages, Leibniz Int'l Proceedings in Informatics (LIPIcs), (2015)

[10] Cachin, C., Guerraoui R., and Rodrigues L., *Introduction to reliable and secure distributed programming*, p. 367, Springer (2012) ISBN 978-3-642-15259-7

[11] Capdevielle C., Johnen C., Milani A. Solo-fast universal constructions for deterministic abortable objects. *Proc. 28th Int'l Symposium on Distributed Computing (DISC14)*, Springer LNCS 8784, pp. 288302 (2014)

[12] Casteigts P., Flocchini P., Godard E., Santoro N., and Yamashita M., Expressivity of time-varying graphs. *Proc. 19th Int'l Symposium on Fundamentals of Computation Theory (FST'13)*, Springer LNCS 8070, pp. 95-106, 2013.

[13] Casteigts A., Flocchini P., Quattrociocchi W., and Santoro N., Time-varying graphs and dynamic networks. *Int'l Journal of Parallel, Emergent and Distributed Systems*, 27(5):387408, 2012.

[14] Chandra T., Hadzilacos V. and Toueg S., The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685-722, 1996.

[15] Chandra T. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225-267 (1996)

[16] Chaudhuri S., More choices allow more faults: set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132-158, 1993.

[17] Cole R. and Vishkin U., Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):32-53 (1986)

[18] Darondeau Ph., Le Guernic P., and Raynal M., Abstract specification of communication systems. *Proc. First IEEE Int'l Conference on Distributed Computing Systems (ICDCS'79)*, IEEEE Press, pp. 339-346, 1979

[19] Delporte-Gallet C., Fauconnier H., Guerraoui R. and Tielmann A. The disagreement power of an adversary. *Distributed Computing*, 24(3-4):137147 (2011)

[20] Dijkstra E.W., Some beautiful arguments using mathematical induction. *Algorithmica*, 13(1):1-8, 1980.

[21] Dolev D., Dwork C., and Stockmeyer L., On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77-97 (1987)

[22] Dwork C., Lynch N., and Stockmeyer L., Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2), 288-323 (1988)

[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] Fraigniaud P., Korman A., and Peleg D., Towards a complexity theory for local distributed computing. *Journal of the ACM*, 60(5), Article 35, 16 pages, 2013.

[25] Friedman R., A. Mostefaoui, S. Rajsbaum, and Raynal M., Distributed agreement problems and their connection with error-correcting codes. *IEEE Transactions on Computers*, 56(7):865-875 (2007)

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

[27] Garey M.R. and Johnson D.S., *Computers and intractability: a guide to the theory of NP-completeness*. W.H. Freeman, New York, 340 pages, 1979.

[28] Guerraoui R. and Lynch N., A general characterization of indulgence. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 3(4): Article 20 (2008)

[29] Guerraoui R. and Raynal M., The Information structure of indulgent consensus. *IEEE Transactions on Computers*, 53(4):453-466 (2004)

[30] Hadzilacos V. Toueg S., A Modular approach to fault-tolerant broadcasts and related problems. *Tech Report 94-1425*, 83 pages, Cornell University (1994)

[31] Hadzilacos V. and Toueg S., On deterministic abortable objects. *Proc. 32nd ACM symposium on Principles of Distributed Computing (PODC'13)*, ACM Press, pp. 4-12 (2013)

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

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

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

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

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

[37] Junqueira F. and Marzullo K., A framework for the design of dependent-failure algorithms. *Concurrency and Computation: Practice and Experience*, 19(17):2255-2269 (2007)

[38] Kuhn F., Lynch N.A., and Oshman R., Distributed computation in dynamic networks. *Proc. 42nd ACM Symposium on Theory of Computing (STOC'10)*, ACM press, pp. 513-522, 2010.

[39] Kuhn F., Moscibroda T., and Wattenhofer R., What cannot be computed locally! *Proc. 23rd ACM Symposium on Principles of Distributed Computing (PODC'04)*, ACM Press, pp. 300-309 (2004)

[40] Kuznetsov P., Understanding non-uniform failure models. *Bulletin of EATCS*, 106(12):54-77 (2012)

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

[42] Lamport L., The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133-169 (1998)

[43] Linial N., Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21(1):193-201 (1992)

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

[45] Lynch N.A., *Distributed algorithms*, p. 872. Morgan Kaufmann (1996)

[46] Moses Y. and Rajsbaum S., A layered analysis of consensus. *SIAM Journal of Computing*, 31:989-1021 (2002)

[47] 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)

[48] 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)

[49] 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, 2016.

[50] Naor M. and Stockmeyer L., What can be computed locally? *SIAM Journal on Computing*, 24(6):1259-1277 (1995)

[51] Padua D. (Ed.), *Encyclopedia of parallel computing*, p. 2180. Springer (2011) ISBN 978-0-387-09765-7

[52] Peleg D., *Distributed computing, a locally sensitive approach*. SIAM Monographs on Discrete Mathematics and Applications, 343 pages (2000) ISBN 0-89871-464-8

[53] Raynal M., *Communication and agreement abstractions for fault-tolerant asynchronous distributed systems*, 251 pages, Morgan & Claypool Pub. (2010) ISBN 978-1-60845-293-4

[54] Raynal M., *Fault-tolerant agreement in synchronous distributed systems*, 167 pages, Morgan & Claypool (2010) ISBN 978-1-608-45525-6

[55] Raynal M., *Concurrent programming: algorithms, principles, and foundations*, 530 pages, Springer (2013) ISBN 978-3-642-32026-2

[56] Raynal M., *Distributed algorithms for message-passing systems*, 515 pages, Springer (2013) ISBN: 978-3-642-38122-5

[57] Raynal M., What can be computed in a distributed system? *From Programs to Systems*, Springer LNCS 8415, pp. 209-224 (2014)

[58] Raynal M., Parallel computing *vs* distributed computing: a great confusion? *Proc. 1st European EUROPAR Workshop on Parallel and Distributed Computing Education (Euro-EDUPAR)*, Springer LNCS 9523, pp. 41-53 (2015)

[59] Raynal M., Messages adversaries. *Encyclopedia of Algorithms*, Springer, 6 pages, (2015) DOI 10.1007/978-3-642-27848-8_609-1

[60] Raynal M., Concurrent systems: hybrid object implementations and abortable objects. *Proc. 21th Int'l European Parallel Computing Conference (EUROPAR'15)*, Springer LNCS 9233, pp. 3-15 (2015)

[61] Raynal M. and Stainer J., Synchrony weakened by message adversaries vs asynchrony restricted by failure detectors. *Proc. 32nd ACM Symposium on Principles of Distributed Computing (PODC '13)*, ACM Press, pp. 166-175, 2013.

[62] Raynal M., Stainer J., and Taubenfeld G., Distributed universality. *Algorithmica* (2015) DOI 10.1007/s00453-015-0053-3

[63] Santoro N. and Widmayer P., Time is not a healer. *Proc. 6th Annual Symposium on Theoretical Aspects of Computer Science (STACS'89)*, Springer LNCS 349, pp. 304-316, 1989.

[64] Santoro N. and Widmayer P., Agreement in synchronous networks with ubiquitous faults. *Theoretical Computer Science*, 384(2-3): 232-249, 2007.

[65] Schmid U., Weiss B., and Keidar I., Impossibility results and lower bounds for consensus under link failures. *SIAM Journal of Computing*, 38(5):1912-1951 (2009)

[66] Suomela J., Survey of local algorithms. *ACM Computing Surveys*, 45(2), Article 24, 40 pages, (2013)

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

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