What Can be Computed in a Distributed System?

Michel Raynal

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

Abstract. Not only the world is distributed, but more and more applications are distributed. Hence, a fundamental question is the following one: What can be computed in a distributed system? The answer to this question depends on the environment in which evolves the considered distributed system, i.e., on the assumptions the system relies on. This environment is very often left implicit and nearly always not formulated in terms of precise underlying requirements. In the extreme case where the environment is such that there is no synchrony assumption and the computing entities may commit failures, many problems become impossible to solve (in these cases, a network of Turing machines where some machines may crash, is less powerful than a single reliable Turing machine). Given a distributed computing problem, it is consequently important to know the weakest assumptions (lower bounds) that give the limits beyond which the considered distributed problem cannot be solved. This paper is a short introduction to this kind of issues. It first presents a few of elements related to distributed computability, and then briefly addresses distributed complexity issues. The style of the paper is voluntarily informal.

Keywords: Agreement, Asynchronous system, Atomicity, Concurrency, Consensus, Crash failure, Distributed complexity, Distributed computability, Distributed computing, Environment, Fault-tolerance, Impossibility, Indulgence, Message adversary, Message-passing system, Progress condition, Read/write system, Synchronous system, Universal construction, Wait-freedom.

1 Definitions

Distributed Computing. Distributed computing was born in the late seventies when researchers and engineers started to take into account the intrinsic characteristic of physically distributed systems [39]. *Distributed computing* arises when one has to solve a problem involving physically distributed entities (called processes, processors, agents, actors, sensors, peers, etc.), such that each entity (a) has only a partial knowledge of the many input parameters of the problem to be solved, and (b) has to compute local outputs which may depend on some non-local input parameters. It follows that the computing entities have necessarily to exchange information and cooperate [52].

Distributed System. A (static) distributed system is made up of n sequential deterministic processes, denoted $p_1, \ldots, p_n$. These processes communicate and synchronize

S. Bensalem et al. (Eds.): FPS 2014 (Sifakis Festschrift), LNCS 8415, pp. 209–224, 2014.

through a communication medium, which is either a network that allows the processes to send and receive messages, or a set of atomic read/write registers (atomic registers could be replaced by "weaker" safe or regular registers, but as shown in [40] –where these registers are defined– safe, regular and atomic registers have the same computational power).

Deterministic means here that the behavior of a process is entirely determined from its initial state, the algorithm it executes, and –according to the communication medium– the sequence of values read from atomic registers or the sequence of received messages (hence, obtaining different sequences of values or receiving messages in a different order can produce different behaviors).

Asynchronous Read/Write or Message-Passing System. In an asynchronous (also called time-free) read/write system, the processes are asynchronous in the sense that, for each of them, there is no assumption on its speed (except that it is positive).

If the communication is by message-passing, the network also is asynchronous, namely, the transfer duration of any message is finite but arbitrary.

Synchronous Message-Passing System. Differently, the main feature of a synchronous system lies in the existence of an upper bound on message transfer delays. Moreover, (a) this bound is known by the processes, and (b) it is assumed that processing durations are negligible with respect to message transfer delays; consequently processing are assumed to have zero duration.

This type of synchrony is abstracted by the notion of round-based computation. The processes proceed in rounds during which each process first sends messages, then, receive messages, and executes local computation. The fundamental assumption which characterizes a synchronous message-passing system is that a message sent during a round is received by its destination process during the very same round.

Process Crash Failure. The most common failure studied in distributed computing is the process crash failure. Such a failure occurs when a process halts unexpectedly. Before crashing it executes correctly its algorithm, and after having crashed, it never recovers.

Let t be the maximal number of processes that may crash; t is a model parameter and the model is called t-resilient model. The asynchronous distributed computing (read/write or message-passing) model in which all processes, except one, may crash is called *wait-free* model. Hence, *wait-free model* is synonym of $(n-1)$-resilient model.

The Notion of Environment and Non-determinism. The *environment* of a distributed system is the set of failures and (a)synchrony patterns in which the system may evolve. Hence, a system does not master its environment but suffers it.

As processes are deterministic, the only non-determinism a distributed system has to cope with is the non-determinism created by its environment.

Complexity vs. Computability Issues. Computability and complexity are the two lenses that allows us to understand and master computing. The following table presents

the main issues encountered in distributed computing, when considering these two lenses.

	Synchronous	Asynchronous
Failure-free	complexity	complexity
Failure-prone	complexity	computability

The rest of this paper illustrates and develops this table. It first addresses computability issues in asynchronous crash-prone distributed systems and presents several ways that have been proposed to circumvent these impossibilities. It then addresses the case of crash-prone synchronous systems, and finally the case of crash-free systems.

2 Are Asynchronous Crash-Prone Distributed Systems Universal?

On the Notion of a Universal Construction. In sequential computing, computability is understood through the Church-Turing's thesis (namely, anything that can be computed, can be computed by a Turing machine). Moreover, when considering the notion of a *universal algorithm* encountered in sequential computing, such an algorithm "has the ability to act like any algorithm whatsoever. It accepts as inputs the description of *any* algorithm A and *any* legal input X, and simply runs, or simulates, A on X. [...] In a sense, a computer [...] is very much like a universal algorithm." [25].

Hence, the question: Is it possible to design a universal algorithm/machine on top of an asynchronous crash-prone distributed system? As we are about to see, it happens that, due the environment (asynchrony and process failures) of a distributed system, and the fact that it cannot control it, distributed computability has a different flavor than computability in sequential computing. Moreover, this is independent of the fact that the communication is by read/write registers or message-passing.

Due to its very nature, distributed computing requires cooperation among the processes. Intuitively, the computability issues come from the fact that, due to the net effect of asynchrony and failures, a process can be unable to know if another process has crashed or is only slow (or equivalently if the channel connecting these processes is slow). Moreover, this is true whatever the individual power of each process. To cite [30], "It follows that the limits of 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".

A Universality Notion for Distributed Computing. A *concurrent* object is an object that can be accessed by several processes. Let us consider a concurrent object Z defined by a sequential specification on a set of total operations. An operation is *total* is, when executed alone, it always returns a result. A specification is sequential, if all the correct behaviors of the object can be described by sequences of operations.

The notion of universality we are interested in concerns the possibility to implement any concurrent object such as Z, despite asynchrony and crashes. If it exists, such an implementation, which takes the sequential specification of Z as input and builds a corresponding concurrent object, is called a *universal construction*. This is depicted in Fig. 1.

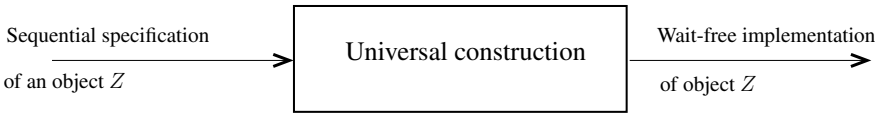

Fig. 1. From a sequential specification to a wait-free implementation

In some cases the object Z encapsulates a service which can be abstracted as a state machine. A replication-based universal construction of such an object Z is usually called a *state machine replication* algorithm [39]. Let us remark that the object Z could also be a Turing machine.

On the Liveness Property Associated with the Constructed Object. Several liveness properties can be associated with the constructed object Z. We consider here *wait-freedom* [27]. The term "wait-freedom" has here a meaning different from the one used in "wait-free model". More precisely, it is here a progress condition for the operations of the constructed objects Z, namely, it states that a universal construction satisfies the "wait-freedom progress condition" if the invocation of an operation on Z by a process can fail to terminate only if the invoking process crashes while executing the operation. We say then (by a slight abuse of language) that the implementation is wait-free. This means that the operation has to terminate even if all the processes, except the invoking process, crash. Let us observe that a wait-free implementation prevents the use of locks (a process that would crash after acquiring a lock could block the system, thereby preventing wait-freedom).

Wait-freedom is the strongest possible progress condition that can be associated with a universal construction (object implementation) in the wait-free model. Other progress conditions suited to the wait-free model are *obstruction-freedom* [29] and *non-blocking* [33]. (See chapter 5 of [51] for a guided tour on progress conditions.)

The Consensus Object. A *consensus* object is a one-shot concurrent object defined by a sequential specification, that provides the processes with a single operation denoted $\mathsf{propose}(v)$ where v is an input parameter (called "proposed value"). "One-shot" means that, given a consensus object, a process invokes at most once the operation $\mathsf{propose}()$. If it terminates, the operation returns a result (called "decided" value). This object can be defined by the three following properties.

- Validity. If a process decides a value, this value has been proposed by a process.
- Agreement. No two processes decide different values.
- Termination. An invocation of $\mathsf{propose}()$ by a process that does not crash terminates.

Consensus-Based Universal Construction Several universal constructions based on atomic registers and consensus objects have been proposed, e.g., [27] (see also chapter 14 of [51]). In that sense, and as depicted in Figure 2, consensus is a universal object to design wait-free universal constructions, i.e., wait-free implementations of any concurrent object defined by a sequential specification. This is depicted in Fig. 2, which completes Fig. 1.

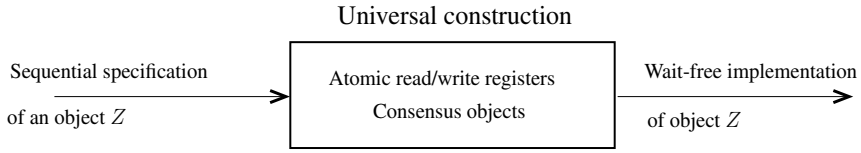

Fig. 2. Universal construction from atomic registers and consensus objects

In a universal construction, consensus objects are used by the processes to build a single total order on the operation invocations applied to the constructed object Z. This is the method used to ensure that the internal representation of Z remains always consistent, and is consequently seen the same way by all processes.

A Fundamental Result. One of the most important of the theoretical results of distributed computing is the celebrated FLP result (named after its authors Fischer, Lynch, and Paterson) [17]), which states that no binary consensus object (a process can only propose $v \in \{0, 1\}$) can be built in an asynchronous message-passing system whose environment states that (even only) one process may crash.

To prove this impossibility result, the authors have introduced the notion of *valence* associated with a global state (also called configuration). Considering binary consensus, a global state is 0-valent (1-valent) if only 0 (1) can be decided from this global state; 0-valent and 1-valent states are univalent states. Otherwise, "the dice are not yet cast", and any of 0 or 1 can be still decided. This is due to the uncontrolled and unpredictable behavior of the environment (i.e., asynchrony and failure pattern of the considered execution). A *decision step* of a construction is one that carries the construction from a bivalent state to a univalent state. The impossibility proof shows that (a) among all possible initial states, there is a bivalent state, and (2) among all possible executions in all possible environments, there is at least one execution that makes the construction always progress from a bivalent state to another bivalent state. It is easy to see that, the impossibility to implement a consensus object is related to the impossibility to break non-determinism (i.e., the impossibility to ensure that, in any execution, there is eventually a transition from a bivalent state to a univalent state).

This message-passing result has then been extended to asynchronous systems in which processes communicate only by reading and writing atomic registers instead of sending and receiving messages [27,42].

Sequential vs Distributed Computing: A Computability Point of View. It follows from the previous impossibility results that a network of Turing machines, that progress asynchronously and where at most one may crash (which are two reasonable assumptions) connected by a message-passing facility, or a read/write shared memory, is computationally less powerful than a single reliable Turing machine. As announced in the first section, this shows that the nature of distributed computability issues is different from the nature of Turing's computability issues, namely, it is not related to the computational power of the individual participants.

The Notion of a Consensus Number of an Object. The notion of *consensus number* of a concurrent object has been introduced by M. Herlihy [27]. The consensus number

of an object X is the largest integer n for which consensus can be be wait-free implemented in a read/write system of n processes enriched with any number of objects X. If there is no largest n, the consensus number is said to be infinite.

It is shown in [27] that the consensus numbers define an infinite hierarchy (hence the name Herlihy's hierarchy, or consensus hierarchy) where we have at levels 1, 2 and $+\infty$:

- Consensus number 1: read/write atomic registers, ...
- Consensus number 2: test&set, swap, fetch&add, stack, queue, ...
- Consensus number $+\infty$: compare&swap, LL/SC, mem-to-mem swap, ...

Universal Objects in the Read/Write Wait-Free System Model. An object is *universal* in the asynchronous read/write wait-free n-process system model, if it allows for the design of a wait-free universal construction in this system model (i.e., an algorithm implementing any concurrent object defined by a sequential specification). It is shown in [27] that any object with consensus number n is universal in a system of n (or less) processes.

From Asynchronous Read/Write Systems to Asynchronous Message-Passing Systems. The previous presentation was focused on asynchronous crash-prone systems in which the processes communicate by reading and writing a shared memory. Hence, the following question: are the impossibility results the same when the processes communicate by sending and receiving messages through a fully connected communication network? This question translates immediately into the following one: While it is easy to simulate the asynchronous wait-free message-passing system model on top of the asynchronous wait-free read/write system model, is the simulation in the other direction possible?

Let us remind that t is a model parameter denoting the maximal number of processes that may crash in an execution. It is shown in [3] that it is not possible to simulate the asynchronous wait-free read/write model on top of the wait-free message-passing model when $t \geq n/2$. This is called the ABD impossibility (named after its authors Attiya, Bar-Noy, Dolev).

The intuition that explains the ABD impossibility can be captured by what is called an *indistinguishability* argument, which relies on the fact that, when $t \geq n/2$, half or more processes may crash in a run. More precisely, as any number of processes may crash, it is possible to construct an execution in which the system "partitions" in two set of processes such that, while there is no crash, the messages between the two partitions take – in both directions– an arbitrarily long time, making the processes in each partition believe that the processes in the other partition have crashed. The system can then progress as two disconnected subsystems.

What is called in some papers [9,20] the "CAP theorem" (where CAP is a shortcut for "Consistency, Availability, Partition-tolerance") can be seen as an impossibility variant combining the ABD and the FLP impossibilities. This theorem, proved in [20], states that, when designing distributed services, it is impossible to design an algorithm that simultaneously ensures the three previous properties.

Playing with Progress Conditions and Consensus Objects. The progress condition called *obstruction-freedom* [29] is weaker than wait-freedom. It states that a process

that invokes an operation on an object is guaranteed to terminate its invocation only when it executes alone for a "long enough period".

The following asymmetric progress condition has been introduced in [35]. An object satisfies (y, x)-*liveness* if it can be accessed by a subset of $y \leq n$ processes only, and wait-freedom is guaranteed for $x \leq y$ processes while obstruction-freedom is guaranteed for the remaining $y - x$ processes. Notice that, (n, n)-liveness is wait-freedom while $(n, 0)$-liveness is obstruction-freedom. Among other results, it is shown in [35] that it is impossible to build a $(n, 1)$-live consensus object from read/write atomic registers and $(n - 1, n - 1)$-live consensus objects. Formulated differently, this states that wait-free consensus objects for $(n-1)$ processes are not computationally strong enough to implement a consensus object for n processes that ensures wait-freedom for only one process (be this process statically or dynamically defined) and the very weak obstruction freedom progress condition for the other processes. Moreover, the paper also shows that (n, x)-live consensus object with $x < n$ has consensus number $x+1$, which thereby establishes a hierarchy for (n, x)-liveness.

3 How to Circumvent Consensus Impossibility

As seen previously, one way to overcome consensus impossibility in a "pure" read/write asynchronous system composed of n processes consists in enriching the system with an object whose consensus number is equal to, or greater than, n. As simulating read/write registers on top of an asynchronous system requires $t < n/2$, this approach does not work for message-passing systems where $t \geq n/2$. Let us also notice that there is no notion of objects with a consensus number in message-passing.

As we are about to see, another approach (which works for both asynchronous read/write systems and asynchronous message-passing systems) consists in enriching the system with an oracle that provides processes with –possibly unreliable– information on failures. This is the failure detector approach.

A third approach consists in restricting the set of possible input vectors (an input vector is a vector –unknown to the processes– whose i-th entry contains the value proposed by process p_i). Hence, instead of enriching the system, this approach considers only a predefined subset of input vectors.

Another type of oracle that can be added to asynchronous systems to circumvent impossibility results, consists in allowing processes to use random numbers.

The Notion of an Unreliable Failure Detector. Failures detectors have been introduced in [12]. From an operational point of view, a failure detector can be seen as a set of n modules, each attached to a process. Failure detectors are divided into classes, according to the particular type of information they give on failures. Different problems (impossible to solve in asynchronous crash-prone systems) may require different classes of failure detectors. Two dimensions can be associated with failure detectors.

– The software engineering dimension of failure detectors. As any type of computer science object (e.g., stack or lock), a failure detector class is defined by a set of abstract properties, i.e., independently of the way these properties are implemented.

Hence, the design and the proof of an algorithm that uses a failure detector are based only on its properties. The implementation of a failure detector is an independent activity. The fact a failure detector can be implemented depends on additional behavioral properties that th environment has to satisfy.

– The ranking dimension of failure detectors. The failure detector approach makes possible the investigation and the statement of the weakest information on failures that allows a given problem to be solved [36]. This permits to rank the difficulty of distributed computing problems, according to the weakest failure detector they need to be solved. If $C1$ and $C2$ are the weakest failure detector classes to solve the problems $Pb1$ and $Pb2$, respectively, and if $C1$ is "strictly stronger" than $C2$ (i.e., gives more information on failures than $C2$), then we say that $Pb1$ is "strictly stronger" than $Pb2$. (The "strictly stronger" notion on failure detectors is a partial order, and consequently some problems cannot be compared with the help of the weakest failure detectors that allow them to be solved).

The Weakest Failure Detector to Solve Consensus in the Wait-Free Read/Write Model. The weakest failure detector class to solve consensus in the wait-free read/write model is denoted Ω. It has been introduced and proved to be minimal in [13]. Ω provides each process p_i with a read-only local variable denoted $leader_i$ that always contains a process identity, and satisfies the following property: there is an unknown but finite time τ after which the variable $leader_i$ of all the non-faulty processes contain the same identity, which is the identity of a non-faulty process.

It is important to notice that (a) there is no way for a process to know if time τ has occurred, and (b) before time τ occurs, there is an anarchy period during which the leader variables can have arbitrary values (e.g., some processes are their own leaders, while the leaders of the others are crashed processes).

Several algorithms implementing Ω, each assuming specific behavioral properties of the underlying system are described in Chapter 6 of [49]. So far, the best algorithm implementing Ω is the one described in [16]. ("Best" means here "with the weakest behavioral assumptions known so far".)

The Notion of an Indulgent Distributed Algorithm. This notion has been introduced in [21], and then investigated for consensus algorithms in [24], and formally characterized in [23]. It is on distributed algorithms that rest on failure detectors.

More precisely, a distributed algorithm is *indulgent* with respect to the failure detector FD it uses to solve a problem Pb if it always guarantees the safety property defining Pb (i.e., whatever the correct/incorrect behavior of FD), and satisfies the liveness property associated with Pb at least when FD behaves correctly. Hence, when the implementation of FD does not satisfies its specification, the algorithm may not terminate, but if it terminates its results are correct.

It is shown in [21] that all the failure detectors defined by an eventual property ("there is a finite time after which ...") are such that the algorithms that use them are indulgent. Ω is such a failure detector. In addition to its theoretical interest, indulgence is very important in practice. This follows from the observation that environments are usually made up of long "stable" periods followed by shorter "unstable" periods. These periods are such that the implementation of an "eventual" failure detector always satisfies its

specification during stable periods, while it does not during unstable periods. Hence, the liveness property of an indulgent algorithm is ensured during the "long enough" stable periods.

The Weakest Failure Detector to Simulate Read/Write on Top of Message-Passing. We have seen that it is possible to simulate an asynchronous crash-prone read/write system on top of an asynchronous crash-prone message-passing system only if $t < n/2$ (let us remind that t denotes the maximal number of processes that may crash in the considered model). As we have seen, intuitively this means that the system cannot partition.

The weakest failure detector to simulate read/write on top of message-passing whatever the value of t, has been introduced in [14]. Denoted Σ, and called *quorum* failure detector, it provides each process p_i with a read-only local variable $quorum_i$, which is a set containing process identities, and is such that (a) the values of any two quorums, each taken at any time, always intersect, and (b) the quorum of any non-faulty process eventually contains only non-faulty processes. The intersection property (a) is a perpetual property, while property (b) is an eventual property.

A simple proof of the minimality of Σ to implement a register on top of an asynchronous message-passing system prone to any number of process crashes can be found in [8]. A generalization of Σ to hybrid communication systems can be found in [34]. "Hybrid" means here that (a) all processes can communicate by sending and receiving messages, and (b) the processes are statically partitioned into clusters and inside each cluster the processes can communicate through read/write registers.

The Weakest Failure Detector to Solve Consensus in the Crash-Prone Message-Passing Model. The weakest failure detector class to solve consensus in the wait free (i.e., when $t = n - 1$) asynchronous message-passing model is the pair (Σ, Ω) [14].

If the system model is such that $t < n/2$, the weakest failure detector class to solve consensus in a message-passing model is Ω [13]. As an exercise, the reader can design a distributed algorithm that builds an atomic register in an asynchronous crash-prone message-passing system model where $t < n/2$ (solutions in [8,14,49]).

Other Weakest Failure Detectors. It is shown in [36] that every distributed computing problem, which can be solved with a failure detector, has a weakest failure detector.

As a simple example, it shown in [26] that the weakest failure detector to solve the *interactive consistency* problem [47] is the *perfect* failure detector defined in [12]. As another example, [15] exhibits the minimum information about failures for solving non-local tasks. (Roughly speaking, "tasks" in distributed computing corresponds to "functions" in sequential computing. "Local" means here that each process can compute its output from its input only.) The weakest failure detector to boost the obstruction-freedom progress condition to wait-freedom is described in [22].

Restricting the Set of Input Vectors for the Consensus Problem. A totally different approach to solve consensus in a read/write system in which up to $t < n$ processes may commit crashes has been proposed in [44]. This approach, called *condition-based*

approach, is related to error-detecting codes [19]. Intuitively an input vector "encodes" a decided value, and the aim of a distributed condition-based consensus algorithm is to "decode" it.

From a more operational point of view, it consists in favoring one of proposed values, while ensuring that this value can be selected by all the processes that decide. To that end, the "favored" value has to appear enough times in the input vector. As a simple example, a condition (set of allowed input vectors) can favor the greatest value present in a vector. To this end, the greatest value in each input vector has to appear at least $t + 1$ times.

Interestingly, the condition-based approach allows to establish a meeting point between computability in crash-prone asynchronous read/write systems and complexity in crash-prone synchronous message-passing systems [45]. Namely, considering systems in which up to t processes may crash, the weakest condition that allows consensus to be solved in an asynchronous read/write system is also the weakest condition that allows consensus to be solved optimally (with respect to the number of rounds) in a synchronous message-passing system.

Breaking Non-determinism with Random Numbers. Let us finally notice that random numbers have been used to solve binary consensus [7]. (There are then algorithms to go from binary consensus to multivalued consensus, both in asynchronous read/write systems [51] and asynchronous message-passing systems [49].) Randomization is used to break the non-determinism which makes the problem impossible to solve without additional computational power.

The termination property of consensus has to be slightly modified to take into account randomization. The corresponding algorithms are round-based, and the termination property becomes: the probability that a non-faulty process has decided by round r tends to 1 when r tends to $+\infty$.

4 Examples of Objects That Can Be Wait-Free Implemented in the Read/Write Wait-Free Model

This section presents two non-trivial objects which can be implemented in the asynchronous read/write wait-free system model (i.e., in which any number of processes may crash). It follows from the previous section that the synchronization and cooperation needed to wait-free implement these objects is relatively weak as it does not require underlying consensus objects. The reader interested in a more global view on objects which can be implemented in read/write systems prone to process crashes can consult [32,51,57].

The Snapshot Object. Snapshot objects have been introduced in [1]. A snapshot object is an array of registers $A[1..n]$, where $A[i]$ can be written only by p_i. It provides the processes with two operations. The first is a write that allows a process p_i to write (only) in $A[i]$. The second operation, denoted snapshot(), can be invoked by any process. It returns the value of the whole array. Moreover, both operations are *atomic*, which means that they appear as if they were executed instantaneously at some point of the time line, each between its start event and its end event [33,40].

A snapshot object can be wait-free implemented in the asynchronous read/write system model where any number of processes may crash (see for example [1,6,43,51]). Hence, it adds no computational power. Its interest lies in the abstraction level (programming comfort) it provides to programmers.

From a complexity point of view, the cost of an implementation of a snapshot object is measured by the number of accesses to basic read/write atomic registers [5]. While the cost of a write into its entry of the array A by a process is 1, the best algorithm designed so far to implement the operation snapshot() is $O(n \log_2 n)$. It is still an open problem to know which is the lower bound associated with the implementation of such an object.

The Renaming Object. This object is a one-shot object which has been introduced in [4], in the context of message-passing systems in which a majority of processes are assumed not to crash. It has then received a lot of attention in the context of read/write wait-free systems (i.e., in systems where any number of processes may crash) [6,11,51].

It is assumed that each process p_i has a name id_i taken from a large name space, whose size is N; the subscript i is then called the index of p_i. Initially a process knows only n and its initial identity id_i. The aim of a renaming object is to allow the processes to obtain new names in a smaller new name space, whose size M is much smaller than N. More precisely, an M-renaming object has a single operation denoted new_name(id) where the input parameter is the identity of the invoking process. An invocation of new_name() returns a new name to the invoking process. More precisely, an M-renaming object is defined by the following properties.

- Validity. A new name is an integer in the set $[1..M]$.
- Agreement. No two processes obtain the same new name.
- Index independence. $\forall\ i, j$, if a process whose index is i obtains the new name v, that process could have obtained the very same new name v if its index had been j.
- Termination. If a process invokes new_name() and does not crash, it eventually obtains a new name.

The index independence property states that, for any process, the new name obtained by that process is independent of its index. This means that, from an operational point of view, the indexes define only an underlying communication infrastructure, i.e., an addressing mechanism that can be used only to access entries of shared arrays. Indexes cannot be used to compute new names. This property prevents a process p_i from choosing i as its new name without any communication.

If only a (non-predetermined) arbitrary subset of processes invoke new_name(), the renaming object is *size-adaptive*. In that case, we have $M = f(p)$ where p is the number of processes which invokes the object operation. On the contrary, if all the processes are assumed to invoke new_name(), the renaming object is not size-adaptive. In that case, $M = f(n)$.

The lower bound on the size of the new name space is $M = 2p - 1$ for size-adaptive renaming. For renaming objects which are not size-adaptive, we have the following. The bound is $M = 2n - 1$ [31], except for an infinity number of values of n for which it is only known that $M \leq 2n - 2$ [10] (this infinite set of values of n includes the values that are not the power of a prime number).

5 On the Complexity Side: A Glance at Synchronous Systems

5.1 The Case of Crash-Prone Synchronous Systems

Synchronous distributed systems do not suffer the same kind of impossibility results as the ones encountered in asynchronous systems. (As a simple example, these systems are computationally strong enough to build a *perfect* failure detector [12].)

The nature of the impossibility results encountered in these systems is similar to the nature of the impossibility results encountered in sequential computing. To illustrate it, let us consider the consensus problem and three types of process failures: crash, send or receive omission, and Byzantine failure.

A process commits a send (receive) omission failure if it "forgets" to send (receive) messages. A process commits a general omission failure if it forgets to send or receive messages. A process commits a Byzantine failure if its behavior does not respect the algorithm it is supposed to execute. Moreover, for the consensus problem to be meaningful in presence of Byzantine failures, its validity and agreement properties have to be restricted as follows. Validity: If all the processes which are not faulty propose the same value, no other value can be decided. Agreement: two non-faulty processes cannot decide different values.

The following upper bounds on the model parameter t are attached to the consensus problem in presence of process failures.

Process failure model	Upper bound on t
crash failure	$t < n$
send omission failure	$t < n$
general omission failure	$t < n/2$
Byzantine failure	$t < n/3$

In all cases, the lower bound on the number of rounds that the processes have to execute is $t + 1$.

5.2 The Case of Crash-Free Synchronous Systems with a Message Adversary

The Notion of a Message Adversary. This notion has been introduced in [55,56] under the name *mobile fault*. A message adversary is a daemon which, at every round, is allowed to suppress messages. Of course, at any round, no process knows in advance which are the links on which messages are suppressed during this round.

If the adversary cannot suppress messages, we have a reliable synchronous system. If, at every round, it suppresses all messages, only local tasks can be computed (and the system is no longer a distributed system). Hence, it is important to characterize an adversary by a property defining its "worst behavior" during each round.

It has been shown in [2] that when the message adversary is constrained by a property denoted TOUR (four tournament), the corresponding synchronous message-passing system model and the asynchronous crash-prone read/write system model have the same computational power for distributed tasks. The adversary TOUR is such that, in each round, and for each pair of processes (p_i, p_j), the adversary is allowed to suppress the

message sent by p_i to p_j or the message sent by p_j to p_i, but not both. More general results connecting synchrony weakened by message adversaries vs asynchrony restricted by failure detectors have been recently established in [53]. An adversary related to message broadcast was introduced in [37].

More generally, the aim of message adversaries is to consider message losses as a normal behavior and not as a faulty behavior from the underlying communication environment. This is strongly related to *dynamic* systems where processes are allowed to move from a location to another location, thereby naturally modifying their neighboring connections (e.g., [54]).

5.3 A Glance at Crash-Free Synchronous Systems with an Arbitrary Network

Crash-Free Synchronous Systems with an Unknown Network. This section discusses briefly synchronous systems in which the communication graph is connected but is not a clique. Moreover, initially a process knows only its neighbors. It is easy to see that if the number of rounds that the processes are allowed to execute is equal to the network diameter, each process can learn all the inputs and then compute its result.

The Notion of Locality in Synchronous Systems. This notion has been introduced in [41], and the associated $\mathcal{LOCAL}$ model has been investigated in [48]. The locality notion has first been used to study complexity issues of distributed algorithms on graphs, and has then addressed more general decision problems.

In a local algorithm, a process is restricted to collect data from other processes which are at distance at most x (i.e., in at most x rounds), where x is smaller than the network diameter.

The main question is then: given a distributed graph problem, is it possible to solve it with a local algorithm? This question is fundamental from a scalability point of view. As a simple example, the optimal vertex coloring problem is not local, while verifying if an arbitrary vertex coloring is such that no two neighbor vertices have the same color can be solved in one round. The interested reader will find in [18,38,41,46,48] complexity results related to locality (associated with problems such as vertex coloring, minimum independent set, minimum vertex cover, etc.).

6 Conclusion

The aim of this paper was to be a short introduction to decidability issues in distributed computing. For more information the reader can consult the following books, some parts of which address distributed computability issues.

- [6,43] are on distributed algorithms in both read/write and send/receive systems where processes can commit failures.
- [51] is on algorithms in *asynchronous* shared memory systems where processes can commit *crash failures*. It focuses on the construction of reliable concurrent objects in the presence of process crashes.
- [57] is on synchronization in shared memory systems and associated complexity bounds.
- [32] is on the design of concurrent objects in shared memory systems.

- [49] is on *asynchronous message-passing* systems where processes are prone to *crash failures*. It presents communication and agreement abstractions for fault-tolerant asynchronous distributed systems. Failure detectors are used to circumvent impossibility results encountered in pure asynchronous systems.
- [50] is on *synchronous* message-passing systems, where the processes are prone to *crash failures, omission failures, or Byzantine failures*. It focuses on the following distributed agreement problems: consensus, interactive consistency, and non-blocking atomic commit.
- [52] is on elementary distributed computing for *failure-free asynchronous message-passing* systems.
- [48] is mainly on the $\mathcal{LOCAL}$ (synchronous) model and associated complexity issues.
- [28] is an introduction to distributed computing based on combinatorial topology.

Acknowledgments. The author wants to thank his colleagues Carole Delporte, Hugues Fauconnier, Eli Gafni, Damien Imbs, Achour Mostéfaoui, Sergio Rajsbaum, Julien Stainer, and Gadi Taubenfeld for stimulating discussions on the nature, the power, and the limits of distributed computing. He wants also to thank François Taïani for constructive comments.

References

1. Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., Shavit, N.: Atomic snapshots of shared memory. Journal of the ACM 40(4), 873–890 (1993)
2. Afek, Y., Gafni, E.: Asynchrony from synchrony. In: Frey, D., Raynal, M., Sarkar, S., Shyamasundar, R.K., Sinha, P. (eds.) ICDCN 2013. LNCS, vol. 7730, pp. 225–239. Springer, Heidelberg (2013)
3. Attiya, H., Bar-Noy, A., Dolev, D.: Sharing memory robustly in message passing systems. Journal of the ACM 42(1), 121–132 (1995)
4. Attiya, H., Bar-Noy, A., Dolev, D., Peleg, D., Reischuk, R.: Renaming in an asynchronous environment. Journal of the ACM 37(3), 524–548 (1990)
5. Attiya, H., Ellen, F., Fatourou, P.: The complexity of updating snapshot objects. Journal of Parallel and Distributed Computing 71(12), 1570–1577 (2010)
6. Attiya, H., Welch, J.L.: Distributed computing: fundamentals, simulations and advanced topics, 2nd edn., 414 pages. Wiley-Interscience (2004) ISBN 0-471-45324-2
7. Ben-Or, M.: Another advantage of free choice: completely asynchronous agreement protocol. In: Proc. 2nd ACM Symposium on Principles of Distributed Computing (PODC 1983), pp. 27–30. ACM Press (1983)
8. Bonnet, F., Raynal, M.: A simple proof of the necessity of the failure detector Σ to implement an atomic register in asynchronous message-passing systems. Information Processing Letters 110(4), 153–157 (2010)
9. Brewer, E.A.: Pushing the CAP: strategies for consistency and availability. IEEE Computer 45(2), 23–29 (2012)
10. Castañeda, A., Rajsbaum, S.: New combinatorial topology bounds for renaming: The upper bound. Journal of the ACM 59(1), Article 3, 49 pages (2012)
11. Castañeda, A., Rajsbaum, S., Raynal, M.: The renaming problem in shared memory systems: an introduction. Elsevier Computer Science Review 5, 229–251 (2011)

12. Chandra, T., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM 43(2), 225–267 (1996)
13. Chandra, T., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. Journal of the ACM 43(4), 685–722 (1996)
14. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R.: Tight failure detection bounds on atomic object implementations. Journal of the ACM 57(4), Article 22 (2010)
15. Delporte-Gallet, C., Fauconnier, H., Toueg, S.: The minimum information about failures for solving non-local tasks in message-passing systems. Distributed Computing 24, 255–269 (2011)
16. Fernández, A., Jiménez, E., Raynal, M., Trédan, G.: A timing assumption and two t-resilient protocols for implementing an eventual leader service in asynchronous shared-memory systems. Algorithmica 56(4), 550–576 (2010)
17. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM 32(2), 374–382 (1985)
18. Fraigniaud, P., Korman, A., Peleg, D.: Towards a complexity theory for local distributed computing. Journal of the ACM 60(5), Article 35, 16 pages (2013)
19. Friedman, R., Mostéfaoui, A., Rajsbaum, S., Raynal, M.: Asynchronous agreement and its relation with error-correcting codes. IEEE Transactions on Computers 56(7), 865–875 (2007)
20. Gilbert, S., Lynch, N.A.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News 33(2), 51–59 (2002)
21. Guerraoui, R.: Indulgent algorithms. In: Proc. 19th ACM Symposium on Principles of Distributed Computing (PODC 2000), pp. 289–298. ACM Press (2000)
22. Guerraoui, G., Kapalka, M., Kouznetsov, P.: The weakest failure detectors to boost obstruction-freedom. Distributed Computing 20(6), 415–433 (2008)
23. Guerraoui, R., Lynch, N.A.: A general characterization of indulgence. ACM Transactions on Autonomous and Adaptive Systems 3(4), Article 20 (2008)
24. Guerraoui, R., Raynal, M.: The information structure of indulgent consensus. IEEE Transactions on Computers 53(4), 453–466 (2004)
25. Harel, D., Feldman, Y.: Algorithmics, the spirit of computing, 572 pages. Springer (2012)
26. Hélary, J.-M., Hurfin, M., Mostéfaoui, A., Raynal, M., Tronel, F.: Computing global functions in asynchronous distributed systems with perfect failure detectors. IEEE Transactions on Parallel and Distributed Systems 11(9), 897–909 (2000)
27. Herlihy, M.P.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems 13(1), 124–149 (1991)
28. Herlihy, M.P., Kozlov, D., Rajsbaum, S.: Distributed computing through combinatorial topology, 336 pages. Morgan Kaufmann/Elsevier (2014) ISBN 9780124045781
29. Herlihy, M.P., Luchangco, V., Moir, M.: Obstruction-free synchronization: double-ended queues as an example. In: Proc. 23rd Int'l IEEE Conference on Distributed Computing Systems (ICDCS 2003), pp. 522–529. IEEE Press (2003)
30. Herlihy, M.P., Rajsbaum, S., Raynal, M.: Power and limits of distributed computing shared memory models. Theoretical Computer Science 509, 3–24 (2013)
31. Herlihy, M.P., Shavit, N.: The topological structure of asynchronous computability. Journal of the ACM 46(6), 858–923 (1999)
32. Herlihy, M.P., Shavit, N.: The art of multiprocessor programming, 508 pages. Morgan Kaufmann (2008) ISBN 978-0-12-370591-4
33. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems 12(3), 463–492 (1990)
34. Imbs, D., Raynal, M.: The weakest failure detector to implement a register in asynchronous systems with hybrid communication. Theoretical Computer Science 512, 130–142 (2013)

35. Imbs, D., Raynal, M., Taubenfeld, G.: On asymmetric progress conditions. In: Proc. 29th ACM Symposium on Principles of Distributed Computing (PODC 2010), pp. 55–64. ACM Press (2010)
36. Jayanti, P., Toueg, S.: Every problem has a weakest failure detector. In: Proc. 27th ACM Symposium on Principles of Distributed Computing (PODC 2008), pp. 75–84. ACM Press (2008)
37. Kuhn, F., Lynch, N.A., Oshman, R.: Distributed computation in dynamic networks. In: Proc. 42nd ACM Symposium on Theory of Computing (STOC 2010), pp. 513–522. ACM Press (2010)
38. Kuhn, F., Moscibroda, T., Wattenhofer, R.: What cannot be computed locally! In: Proc. 23rd ACM Symposium on Principles of Distributed Computing (PODC 2004), pp. 300–309. ACM Press (2004)
39. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM 21(7), 558–565 (1978)
40. Lamport, L.: On inter-process communications, Part I: Basic formalism. Distributed Computing 1(2), 77–85 (1986)
41. Linial, N.: Locality in distributed graph algorithms. SIAM Journal on Computing 21(1), 193–201 (1992)
42. Loui, M., Abu-Amara, H.: Memory requirements for agreement among unreliable asynchronous processes. Advances in Computing Research 4, 163–183 (1987)
43. Lynch, N.A.: Distributed algorithms, 872 pages. Morgan Kaufmann (1996)
44. Mostéfaoui, A., Rajsbaum, S., Raynal, M.: Conditions on input vectors for consensus solvability in asynchronous distributed systems. Journal of the ACM 50(6), 922–954 (2003)
45. Mostéfaoui, A., Rajsbaum, S., Raynal, M.: Synchronous condition-based consensus. Distributed Computing 18(5), 325–343 (2006)
46. Naor, M., Stockmeyer, L.: What can be computed locally? In: Proc. 25th ACM Symposium on Theory of Computing (STOC 1993), pp. 184–193. ACM Press (1993)
47. Pease, M., Shostak, R., Lamport, L.: Reaching agreement in the presence of faults. Journal of the ACM 27, 228–234 (1980)
48. Peleg, D.: Distributed computing, a locally sensitive approach. SIAM Monographs on Discrete Mathematics and Applications, 343 pages (2000) ISBN 0-89871-464-8
49. Raynal, M.: Communication and agreement abstractions for fault-tolerant asynchronous distributed systems, 251 pages. Morgan & Claypool Pub. (2010) ISBN 978-1-60845-293-4
50. Raynal, M.: Fault-tolerant agreement in synchronous message-passing systems, 165 pages. Morgan & Claypool Publishers (2010) ISBN 978-1-60845-525-6
51. Raynal, M.: Concurrent programming: algorithms, principles, and foundations, 530 pages. Springer (2013) ISBN 978-3-642-32026-2
52. Raynal, M.: Distributed algorithms for message-passing systems, 515 pages. Springer, ISBN: 978-3-642-38122-5
53. Raynal, M., Stainer, J.: Round-based synchrony weakened by message adversaries *vs* asynchrony enriched with failure detectors. In: Proc. 33rd ACM Symposium on Principles of Distributed Computing (PODC 2013), pp. 166–175. ACM Press (2013)
54. Raynal, M., Stainer, J., Cao, J., Wu, W.: A simple broadcast algorithm for recurrent dynamic systems. In: Proc. 28th IEEE Int'l Conference on Advanced Information Networking and Applications (AINA 2014), 8 pages. IEEE Press (2014)
55. Santoro, N., Widmayer, P.: Time is not a healer. In: Cori, R., Monien, B. (eds.) STACS 1989. LNCS, vol. 349, pp. 304–316. Springer, Heidelberg (1989)
56. Santoro, N., Widmayer, P.: Agreement in synchronous networks with ubiquitous faults. Theoretical Computer Science 384(2-3), 232–249 (2007)
57. Taubenfeld, G.: Synchronization algorithms and concurrent programming, 423 pages. Pearson Education/Prentice Hall (2006) ISBN 0-131-97259-6