Distributed Computability: A Few Results Masters Students Should Know

Michel Raynal

Distributed Computability: a Few Results Masters Students Should Know

Michel Raynal

Univ Rennes, IRISA, Inria, CNRS, 35042 Rennes, France
Department of Computing, Hong Kong Polytechnic University

Abstract

As today Informatics is more and more (~~driven~~) eaten by its applications, it becomes more and more important to know what can be done and what cannot be done. For a long time now, this is well known in sequential computing. But today the world becomes more and more distributed, and consequently more and more applications are distributed. As a result, it becomes important, or even crucial, to understand what is distributed computing and which are its power and its limits. This article is a step in this direction when looking from an agreement-oriented fault-tolerance point of view.

Teaching is not an accumulation of facts.
L. Lamport [56]

Correctness may be theoretical, ... but incorrectness has practical impact.
M. Herlihy

- Don't you have a more recent newspaper? I've known these news for two days.
- Read them again. In a few days they will be new again.
La Marca del viento (2019), Eduardo Fernando Valera

1 Introduction

1.1 Sequential and parallel computing

Sequential computing All curricula in informatics have a section devoted to computability in sequential computing. It follows that students know fundamental results such as

- the automata hierarchy, namely: Finite State automata $\subsetneq$ Pushdown automata $\subsetneq$ Turing machine,
- the Church-Turing *thesis* stating that everything that can be mechanically computed can be computed by a Turing machine,
- the impossibility to solve some well-defined problems (such that the halting problem [90]),
- The impossibility to bypass some bounds (for example design a comparison-based sorting algorithm whose number of comparisons is always smaller than $O(n \log n)$, where n is the number of elements to sort).
- The notion of non-determinism and the P $\neq$ NP conjecture.

This basic theoretical knowledge, together with algorithms and programming, constitutes the pillars on which stands sequential computing.

Parallel computing Parallel computing was introduced to solve *efficiency* issues [92]. As in the recursive thinking, the idea is to decompose a problem Pb in independent sub-problems, and then recompose their solutions to obtain a solution to Pb. Said in an other way, the key of parallel computing is *data independence*. It consists in making data (or blocs of data) independent of each other, so that they can be processed independently the ones from the others –i.e. without interactions–, which, from an efficiency point of view, means processing them simultaneously.

The decomposition of the set of input data in independent data is under the control of the programmer and constitutes the core of the design of parallel algorithms. Put it differently, if efficiency was not the issue, the problems addressed in parallel computing could be solved with a sequential algorithm.

1.2 Distributed computing

An informal view (from [77]) Distributed computing was born in the late 1970s when researchers and practitioners started taking into account the intrinsic characteristic of physically distributed systems. The field then emerged as a specialized research area distinct from networking, operating systems, and parallel computing.

Distributed computing arises when one has to solve a problem in terms of pre-defined 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. While parallel computing and real-time computing can be characterized, respectively, by the terms *efficiency* and *on-time computing*, distributed computing can be characterized by the term *coordination in the presence of uncertainty*. In distributed computing, the computing entities are not defined by the programmer but *imposed* to her, who has to allow them to correctly *cooperate*.

This uncertainty is created by asynchrony, multiplicity of control flows, absence of shared memory and global time, failure, dynamicity, mobility, etc., all called *adversaries*. The set of of the adversaries defines what is called the *environment*. Let us notice that the environment constitutes an implicit and unknown input of any distributed execution (in the sense it can affect the result produced by the execution).

Mastering one form or another of uncertainty is pervasive in all distributed computing problems. A main difficulty in designing distributed algorithms comes from the fact that no entity cooperating in the achievement of a common goal can have an instantaneous knowledge of the current state of the other entities, it can only know some of their past local states.

As we can see, the aim of parallel computing and distributed computing can be seen as being orthogonal: parallel computing is looking for data independence in order to split the computation in independent computing entities to obtain efficient algorithms, while the aim of distributed computing is to allow computing entities to cooperate in the presence of adversaries.

Two quotations Here are two quotes that can help the reader capture the spirit of distributed computing. The first one is an humorous quote from L. Lamport [54]:

> A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

This quote can be seen as a funny version of the FLP theorem (presented below). The second one (from [42]) stresses the fundamental difference between sequential computing and distributed computing when looking at computability.

> In sequential systems, computability is understood through the Church-Turing Thesis: anything that can be computed, can be computed by a Turing Machine. In distributed systems, where computations require coordination among multiple participants, computability questions have a different flavor. Here, too, there are many problems which are not computable,

but these limits to computability reflect the difficulty of making decisions in the face of ambiguity, and have little to do with the inherent computational power of individual participants.

2 Preliminaries

2.1 Computing models

The reader interested in the notion of *model* can consult [85].

Computing entities In the following we consider a static system is made up of a set of n asynchronous computing entities (called processes in the following), denoted p_1, ..., p_n. The local power of a process is the one of a Turing machine. "Asynchronous" means that each process progresses at its own speed, which can vary with time, and remains always unknown to the other processes.

Process failures Two types of process failures are considered.

- Crash failure model. A crash is a premature unanticipated halt by a process. Moreover once crashed, a process remains crashed forever. It is important to notice that a process behaves correctly until it crashes (if it ever crashes).
- Byzantine failure model. This type of failure was introduced in the early eighties [57, 70]. A Byzantine process is a process that does not follow the behavior defined by the algorithm it is assumed to execute. Such an incorrect behavior can be intentional or not. When intentional, it is sometimes called "malicious". In this model a crash is considered as a Byzantine failure.

Given a run and according to the failure model, a process that does not commit a failure is a *correct* (or *non-faulty*) process. Otherwise it is a *faulty* process. The fact we do not know in advance which are the correct and the faulty processes is one of main sources of non-determinism that distributed algorithms have to master.

Let us also notice that these are more sophisticated Byzantine failure models –not addressed in this article– that distinguish crash failures from Byzantine failures (the former being less severe than the latter).

Communication model We consider two communication models.

- Shared memory model. In this model the processes communicate through atomic registers. According to the primitive operations the processes can use to access these registers, two sub-models can be defined.
  - In the basic read/write (RW) model, the only operations a process can invoke to access a register are read and write.
  - In the read/modify/write (RMW) model, in addition to read and write primitive operations, a process can invoke one or more sophisticated primitive operations such as Test&Set(), Fetch&Add(), Compare&Swap(), or LL/SC (linked load/store conditional). According to the operations enriching the basic RW model, there are many RMW models. The model enriched with the primitive operation op is denoted RMW+op.

    As an example let us consider the model RMW+Compare&Swap. As LL/SC, the operation Compare&Swap, is an atomic conditional write. The effect of Compare&Swap(R, old, new) where R is a shared register and old and new are two values is the following. If $R = old$, the operation assigns new to R and returns `true`. If $R \neq old$, R is not modified and the operation returns `false`.

- Message-passing model. In this model the processes exchange message on top of a communication network. In this case, in addition to the local power provided by a Turing machine, the processes can send and receive messages. It is assumed that the network is reliable (i.e., there is neither loss, creation, nor alteration of messages).

2.2 Progress conditions

Case of the failure-free model In a failure-free context, two progress conditions have been defined for the high level operations invoked on shared objects defined by the programmer or supplied by a library (e.g., the operations push() and pop() associated with a stack). According to the underlying communication model, these objects are implemented on top of registers of the shared memory, or on top of the local memories of the processes (in this case, the communication medium can be message-passing or shared memory). *Deadlock-freedom* states that if one or more processes invoke an high level operation on a shared object, at least one invocation will succeed. *Starvation-freedom* is a stronger progress condition, namely it states that if any process invokes an operation on a shared object, its invocation will succeed. Deadlock-free and starvation-free mutual exclusions are a classical techniques used to protect accesses to shared objects [73].

Case of the shared memory crash-prone model In this context, the situation is different. The crash of a process p while it is accessing a shared object can prevent other processes from forever accessing the object. Intuitively, this is due to the fact that no process can know if p crashed or is is only very slow. As a consequence, since the early nineties, progress conditions suited to asynchronous crash-prone systems have been proposed.

- Obstruction-freedom [41]. An object operation is *obstruction-free* if, assuming the invoking process p does not crash and executes in isolation for a long enough period, its invocation terminates. "In isolation" means that, if any, the concurrent invocations of operations on the same object by other processes have stopped their execution -maybe in the middle of their code– and remain pending until p returns from its invocation).
- Non-blocking [45]. An object operation is *non-blocking* if, whatever the number of concurrent invocations, at least one of them terminates. Non-blocking extends deadlock-freedom to process crash failures.
- Wait-freedom [40]. An object operation is *wait-free* if all its invocations by processes that do not crash terminate. Wait-freedom extends starvation-freedom to process crash-failures.

In the case of Byzantine failures, the progress condition states that the invocation of an operation by a correct process must terminate if the number of Byzantine processes doe not bypass a pre-defined threshold.

Case of the asynchronous message-passing model In this context, according to the problem (object to implement) that is solved, the previous progress conditions may apply in the case of crash failures.

But this not true for all the problems. As for Byzantine failures in shared memory systems, in many cases, the progress condition that is considered states that an operation invoked by a correct process must terminate if some assumptions are satisfied. These assumptions usually involve the maximum number of processes that can be faulty [43].

3 Asynchronous Crash-Prone Shared Memory

3.1 The consensus object

A notion of universality Considering the asynchronous crash-prone shared memory model, M. Herlihy introduced the notions of a *universal object* and a *universal construction* [40]. This notion of universality concerns the *wait-free implementation* of the objects defined by a sequential specification ("wait-free implementation" means that all the operations of the object that is built must be wait-free). To this end, he introduced the notion of a *universal object type*. An object type T is universal if a universal construction can be built from

- any number of objects of type T, and
- any number of atomic read/write registers.

It is shown in [40] that the *consensus* object type (defined below) is universal.

The Consensus object Consensus is a one-shot object that provides the processes with a single operation denoted propose(). (One-shot" means that a process may invoke d propose() at most once). When a process p_i invokes propose(v_i) we say " p_i proposes the value v_i". Such an invocation returns a value v and, when this occurs, we say "p_i decides v". In the context of process crah failures, *consensus* is defined by the three following properties (validity and agreement are safety properties, while wait-freedom is a liveness property).

- Validity: The value decided by a process is a proposed value.
- Agreement: No two processes decide different values.
- Wait-freedom: The invocation of propose() by a non-faulty process terminates.

Principles underlying universal constructions The universality of the consensus object comes from the fact that it allows the processes to *agree on a unique order* to apply the operations invoked on the object O that is built. On the client side each process can invoke an operation on O by announcing it to all the processes. On the server side, the processes execute a sequence of rounds, each using a new consensus object. During each round, each process proposes to the consensus object the operations it sees as being proposed and not yet applied to O. The proposal output by the consensus object is the winner proposal for the corresponding round, and the sequence of operations it contains is applied by each process to its local copy of the object (if the state machine replication –SMR– paradigm is used [53, 78, 84]), or to the unique copy of the object in the case where the object O is stored in a shared memory [40]. All the universal constructions rely on a helping mechanism [19], which ensures that the operations invoked by processes that crash during their invocations are applied by all or none the non-crashed processes.

More advanced topic: Yet more universal constructions The previous constructions consider the construction of a single object. A more general construction is described in [36], that considers the case where, not only one, but k objects are built and the wait-freedom property is satisfied for at least one of them. This construction relies on k-set agreement, a generalization of consensus [22].

A yet more general construction is described in [81], that considers the simultaneous construction of k objects such that the wait-freedom property is satisfied for at least ℓ of them, $1 \leq \ell \leq k$. Such a construction relies on the k-simultaneous consensus object defined in [4].

3.2 The consensus hierarchy

The consensus number notion This notion was introduced in [40]. The consensus number of an object type T is the greatest integer n such that, despite asynchrony and any number of process crashes, it is possible to build a consensus object from atomic read/write registers and objects of type T in a system of n processes. If there is no such finite n, the consensus number of T is $+\infty$.

It is shown in [40] that the consensus numbers defines an infinite hierarchy such that

- Read/write registers have consensus number 1. (The first proof showing that consensus cannot be wait-free implemented on top of read/write registers in the presence of asynchrony and any number of process crashes appeared in [58].)
- Test&Set registers, Fetch&Add registers, stacks, queues have consensus number 2.
- Compare&Swap registers, LL/SC (linked load/store conditional) registers, Memory-to-Memory swap registers have consensus number $+\infty$.

A simple object family covering the full hierarchy The *k-sliding read/write register* object, in short RW(k) was introduced in [62] (a similar object was independently introduced in [32]). This object is a a natural generalization of an atomic read/write register, which corresponds to the case $k = 1$). This object is an (initially empty) sequence of values such that the write of a value v adds v at the end of the sequence, and a read returns the sub-sequence of the last k written values (if only $x < k$ values have been written, the read return these x values). It is easy to see that the type RW(1)is the read/write register type, while RW($+\infty$) is nothing else than a ledger [77].

Universal constructions Universal constructions based on consensus objects or on operations whose consensus number is $+\infty$ (such as LL/SC), are described in many articles, e.g. [31, 33, 40, 74, 76], and and in textbooks such as [10, 44, 74, 87].

Objects at level 1 of the hierarchy It follows from the previous hierarchy that, as they have consensus number 2, objects such as stacks and queues cannot be wait-free implemented on top of read/write registers only. Differently, important objects such as snapshot [1, 6, 49], renaming [8, 18], immediate snapshot [13], and approximate agreement [26] have consensus number 1 and can consequently be wait-free built on top of read/write registers only, despite asynchrony and any number of process crashes. The interested reader can consult textbooks such as [44, 74, 87] where are described wait-free implementation of such objects. The family of the CRDT objects (Conflict-free Replicated Data Types [86]) is a sub-class of the objects at level 1 of the consensus hierarchy.

Implication of the hierarchy for consensus number > 1 It follows from Herlihy's infinite hierarchy, that no object with consensus number $x > 1$ can be wait-free implemented on top of an object whose consensus number is smaller than x.

It follows from this observation that, while a shared stack (or a queue) can be implemented on top of atomic read/write atomic registers in an asynchronous failure-free systems (whatever the number of processes), this is impossible in a crash-prone system. This, which at first glance may seem surprising and counter-intuitive, is due to the fact their consensus number is 2, while the consensus number of read/write registers is 1.

It follows that, differently from what occurs in sequential computing (where read/write registers are the cells of a Turing machine) and more generally in failure-free concurrent computing, read/write registers are not universal in asynchronous crash-prone systems, where consensus is needed to cope with the non-determinism created by the environment (net effect of asynchrony and crash failures).

Read/write with respect to read/modify/write An important point lies in the fact that a write operation not only defines a new value for the register R to which it is applied, but also erases its past which is then lost forever. Differently, while operations such as $\mathsf{Fetch\&Add}(R)$ or $\mathsf{Test\&Set}(R)$ (whose consensus number is 2) are unconditional writes, they return the previous value of the register R to which they are applied (and consequently this value is not lost forever and can used by the invoking process). More, the operation $\mathsf{Compare\&Swap}(R, old, new)$ (whose consensus number is $+\infty$) is a conditional write (it modifies the register R to which it is applied only if its current value is equal to the parameter value old).

Domain of consensus numbers The notion of consensus number has been defined in the context of the asynchronous shared memory model in which any number of processes may crash. Such a notion cannot be directly applied to message-passing systems or Byzantine failures.

Advanced topics The multiplicative power of consensus numbers has been investigated in [47]. An extension of the consensus number hierarchy to multi-threaded systems is presented in [71]. It has been shown in [2, 25] that given any integer $x \geq 1$, there is an infinite number of objects $O(x, 1)$, $O(x, 2)$, etc., such that, while they all have consensus number x, it is possible to build $O(x, y)$ from $O(x, y-1)$ but $O(x, y-1)$ cannot be built from $O(x, y)$ (hence, there is an infinite computability-related sub-hierarchy between any two consecutive consensus numbers). An exhaustive survey of the consensus number land is presented in [79].

A notion close but different from consensus numbers, called d-solo executions, is presented in [43]. A process runs solo when it computes its local output without receiving any information from other processes, either because they crashed or they are too slow. While in wait-free shared memory models at most one process may run solo in an execution, any number of processes may have to run solo in an asynchronous wait-free message-passing model. In the *d-solo* model, $1 \leq d \leq n$, up to d processes may run solo. A hierarchy of wait-free models is described in [43] that, while weaker than the basic crash-prone read/write model, are nevertheless strong enough to solve non-trivial tasks.

4 Synchronous Message-Passing Systems

From now on, we assume the inter-process communication is through message-passing. Moreover, each pair of processes is connected by a reliable bidirectional channel.

4.1 Round-based communication

In a synchronous message-passing system the processes execute a sequence of rounds whose progress is governed by an external clock, the progress of which defines a sequence of *rounds*. Each round r is composed three phases.

- Phase 1: at the beginning of a round r each process sends a message to the other processes. If a process crashes during this sending phase, an arbitrary subset of processes receives the message.
- Phase 2: a process receives messages sent by the other processes.
- Phase 3: according to its current local state and the messages it has received, a process does local computations that modifies its local state.

The synchrony property The fundamental synchrony property (provided for free) of the synchronous message-passing system model lies in the fact that a message sent by a process during a round r is received by its destination process during the very same round r.

If follows from the synchrony property that, if a process p crashes during round r, a non-crashed processes learns it during round r or round $(r+1)$ (because it does not receive a message from p). As we can see, this synchrony property drastically reduces the non-determinism due to environment.

4.2 Consensus in the round-based synchronous communication model

Let t, $1 \leq t < n$, be a model parameter denoting the maximum number of processes that may be faulty in a run. It is assumed that t is known by the processes. Moreover, given a run, let f, $0 \leq f \leq t$ be the number of faulty processes in this run. We have the following results concerning consensus [5, 27, 34].

The case of process crashes In the synchronous failure-prone message-passing model, the synchrony assumption is strong enough to allow consensus to be solved. More precisely, we have the following.

- Consensus can be solved for any value of $t < n$.
- The maximal number of rounds to solve consensus is $\mathsf{min}(f+2, t+1)$.

The case of Byzantine processes As a Byzantine process can never decide or can decide any value, and the non-Byzantine processes can propose different values, the definition of consensus must be appropriately modified in order an algorithm can be designed. We consider here the following classical definition [10, 27, 77].

- Validity: If all the non-faulty processes propose the same value, this value is decided. In the other cases any value can be decided.
- Agreement: No two non-faulty processes decide different values.
- Wait-freedom: The invocation of $\mathsf{propose}()$ by a non-faulty process terminates.

Let us observe that, in the case of binary consensus (only two values can be proposed), it follows from the validity property that the value decided by the non-faulty processes is always a value proposed by one of them [77].

A stronger validity property is used in some articles, namely: If $(t+1)$ non-faulty processes propose the same value, this value is decided. In the other cases any value can be decided. This does not change the following results.

Considering the round-based communication model, we have the following results.

- Consensus can be solved if and only if $t < n/3$ [57, 70].
- The maximal number of rounds needed to solve Byzantine consensus is $\mathsf{min}(f+2, t+1)$ [27].
- If the model is enriched with message authentication, the previous necessary and sufficient condition ($t < n/3$) becomes $t < n/2$.

Remark 1 Let us notice that there are algorithms that (at the additional cost of two synchronous rounds) reduce multi-valued synchronous consensus to binary synchronous consensus in the presence of $t < n/3$ Byzantine processes [66, 77, 91].

Remark 2 As far as synchrony assumptions are concerned, it is important to say that the synchrony offered by the round-based synchronous communication model is not the weakest synchrony assumption that allows consensus to be solved. In other words, the synchronous communication model is stronger than necessary. This point is addressed in Section 5.4 for the case of Byzantine failures.

5 Asynchronous Message-Passing Systems

An asynchronous message-passing distributed system is such that both the processes and the communication channels are asynchronous. An asynchronous channel is such that there is no assumption of the transfer delay of each message, except it is finite.

In the following we consider three problems, and for each of them consider the case of crash failures an the case of Byzantine processes. As previously, t denotes the maximal number of processes that may commit failures.

5.1 Construction of an atomic read/write register

As a read/write register is the most basic object of sequential computing, and, more generally, a file is nothing else than a "big" read/write register, it is natural to try to build a such a register on top of an synchronous message-passing system prone to process (crash or Byzantine) failures.

The case of process crashes The most important result, due to [7], is the following necessary and sufficient condition:

- $t < n/2$ is a necessary and sufficient condition to build an atomic read/write register in asynchronous n-process system prone to process crash failures.

Intuitively, this is due to the following reason. If $t \geq n/2$, the processes can be partitioned in two subsets $Q1$ and $Q2$ such that the communication inside each partition is rapid while the communication between the two partitions is arbitrarily slow. In this case, the processes of $Q1$ (reps. $Q2$) cannot distinguish the case were the processes $Q2$ (reps. $Q1$) have crashed or are arbitrarily slow. This is called an *indistinguishability argument* [9]. Differently, if $t < n/2$, it is possible for a process to communicate with a majority of processes without risking to become blocked forever. As any two majorities intersect, we have then $Q1 \cap Q2 \neq \emptyset$, from which portioning can be prevented.

The efficiency of algorithms implementing read/write registers in the presence of asynchrony and a majority of processes that do not crash has been addressed in [29, 68]. It is shown in [66] that, in addition to the value of the register, messages have to carry two bits of control information.

The case of Byzantine processes As a Byzantine process can pollute each register it writes, the shared memory must be restricted to be an array $M[1..n]$ of single-writer multi-reader (SWMR) atomic registers. Hence, $M[i]$ can read by all the processes but written only by p_i. Considering such a context, the following necessary and sufficient condition is proved in [46]:

- $t < n/3$ is a necessary and sufficient condition to build a shared memory made up SWMR atomic registers in an asynchronous n-process system in which up to t process may be Byzantine.

Algorithms building such a Byzantine-tolerant shared memory are presented in [46, 63].

An intuitive explanation of the $t < n/3$ assumption Let us consider the worst case, namely $n = 3t+1$. So, there are at least $n-t = 2t+1$ non-faulty processes. Let us partition the processes in four sets: $Q1$, $Q2$, $Q3$, and $Q4$, so that each of $Q1$, $Q2$, and $Q3$ contains t processes, $Q4$ contains a single process, and all the up to t Byzantine processes are in $Q3$. As there are $n - t = 2t + 1$ non-faulty processes, a process can communicate with this number of processes without risking to be blocked. But, due to asynchrony, it is possible that, inside the set of $n - t = 2t + 1$ processes with which it communicates, t processes are Byzantine, each of them sending different values to distinct processes (while it is assumed to send the same value). Given any two non-faulty processes p_i and p_j, let $P_i = Q1 \cup Q3 \cup Q_4$ the set of processes the set of processes communicating with p_i, and $P_j = Q2 \cup Q3 \cup Q_4$. We have then

$|P_i \cap P_j| = t + 1$, when means that there is at least one non-faulty that communicate with both p_i and p_j.

5.2 Consensus in the presence of asynchrony and process crashes

Consensus impossibility One of most important results of asynchronous crash-prone message-passing rums i the following one, known under the name FLP, according to its the name of its authors (Fischer, Lynch, and Paterson [35]).

- There is no deterministic algorithm that solves consensus in asynchronous message-passing systems in which even a single process may crash.

To prove their theorem, the authors considered binary consensus and, assuming an algorithm can solve consensus, they introduced a new notion that revealed to be extremely simple and powerful, namely the notion of *valence* of a global state (also called configuration) of the algorithm assumed to solve consensus.

A state in 0-valent (resp.,1-valent), if only 0 (resp. 1) can be decided from this state (maybe no one knows it, but in such a state "the dices are thrown"). 0-valent states and 1-valent states are called mono-valent states. A bi-valent state in a state in which nothing has yet been decided (maybe no one knows it, but in such a state "the dices are not yet thrown").

The proof is then made up of two parts. The first consists in showing that among all possible global states, at least one of them is bi-valent. The second part consists in showing, that given any algorithm assumed to solve consensus, if it is in a bi-valent state there is a execution of it that moves it in a new bi-valent state, from which follows that the algorithm will never terminate.

How to circumvent FLP? Several approaches have proposed to circumvent the impossibility of consensus in asynchronous crash-prone systems.

- Add scheduling assumptions that restrict the set of possible behaviors (e.g., [16, 28, 30, 61]).
- Provide the processes with information on failures [21]. It has been shown in [20] that the weakest information on crash failures that allows consensus to be solved is captured by the failure detector, called *eventual leader* and denoted Ω [20]. Such a failure detector provides each process p_i with a read-only variable $leader_i$, which always contains a process identity, and these local variables are such that there is an unknown but finite time τ after which they all contain the same identity, which is the identity of a process that does not crash.
- Restriction on the sets of input vectors. An input vector is the vector (unknown by the processes) of size n, the entry i containing the value proposed by p_i. This approach, called condition-based, was introduced in [64].
- Use randomization. For crash failures, this approach was introduced in [11].

Several algorithms based on each of the previous assumptions are described in [77].

Advanced topic Computability-related equivalences between round-based synchronous models constrained by message adversaries and and asynchronous crash-prone message-passing models enriched with failure detectors are presented in [3, 80].

5.3 Consensus in the presence of asynchrony and Byzantine processes

As Byzantine failures are more severe than crash failures The FLP impossibility remains valid in the Byzantine failure model. Many Byzantine-tolerant consensus algorithms have been proposed. They are based on asynchronous rounds, which –differently from the synchronous model– must be explicitly built by the algorithm. The major part of these algorithms consider binary consensus.

Starting with an early article by M. Rabin [72], these algorithms use randomization to circumvent the consensus impossibility. Among the existing algorithms the one described in [60] is optimal in several respects. More precisely:

- It requires $t < n/3$ and is consequently optimal with respect to t.
- It uses a constant number of communication steps per round.
- The expected number of rounds to decide is constant.
- The message complexity is $O(n^2)$ messages per round.
- Each message carries its type, a round number plus a constant number of bits.
- The algorithm uses a weak common coin. Weak means here that there is a constant probability that, at every round, the coin returns different values to distinct processes.
- Finally, the algorithm does not assume a computationally limited adversary (and consequently it does not rely on signed messages).

Algorithms that reduce Byzantine multi-valued consensus to Byzantine binary consensus are described in [12, 65, 67].

Remark The necessary and sufficient condition $t < n/3$ can be weakened into $t < n/2$ if the underlying asynchronous architecture is enriched with an appropriate temperproof distributed component such as TTCB (Trusted Timely Computing Base). As an example, the interested reader will find in [23, 24, 83] the development of such an approach.

5.4 Byzantine consensus: the minimal synchrony assumption

While consensus can be solved in the round-based synchronous message-passing system model where up to $t < n/3$ processes can be Byzantine, it cannot be solved in fully asynchronous process model where up to $t < n/3$ processes can be Byzantine. Hence, the question: which is the weakest synchrony assumption that allows consensus to be solved in the presence of up to $t < n/3$ Byzantine processes. This question has been answered in [14], where it is shown that the following synchrony assumption is necessary and sufficient condition.

Uni-directional channels Let us assume that each pair of processes is connected by two uni-directional communication channels. (This is to be as general as possible as it becomes possible to associate different transfer delays to each direction of a bi-directional channel.)

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

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

It is shown in [14] that the existence of an eventual $\langle t+1 \rangle$bisource is the weakest synchrony assumption that allows Byzantine-tolerant consensus to be solved. This article presents also an algorithm based on this assumption.

6 Reliable Broadcast

Reliable broadcast belongs to the family of fundamental problems of fault-tolerant distributed computing [15, 16]. As previously, it comes in two versions according to the process failure model. *Reliable* means here that the set of the messages that are delivered satisfy well-defined properties, which are crucial to the design of provably correct distributed software.

Reliable broadcast in the presence of crash failures This communication abstraction, that provides the processes with two operations denoted R_broadcast() and R_deliver(), is defined by the following properties. When a process invokes R_broadcast(m), we say that it "r-broadcasts m". Similarly, when it issues R_deliver() and obtains the message m, we say that it "r-delivers m".

- Validity. If a process p_i r-delivers a message m from a process p_j, m has previously been r-broadcast by p_j.
- Integrity. Assuming all the messages are different, no process r-delivers twice the same message (no duplication).
- Termination-1. If a non-faulty process r-broadcasts a message m, it r-delivers m.
- Termination-2. If a process r-delivers a message m, every non-faulty process r-delivers the message m.

Let us notice that in the last item, the process that r-delivers m can be faulty or non-faulty. It is easy to see that this communication abstraction satisfies the following properties.

- The non-faulty processes r-deliver the same set M of messages, which includes all the the messages they r-broadcast and a subset of the messages r-broadcast by faulty processes.
- A faulty process r-delivers a subset of the messages in M.

This abstraction is easy to implement. When a process invokes R_broadcast(m), it sends a copy of m to each process, and when a process receives a message m for the first time it forwards it to the other processes (except its sender). It is easy to see that an invocation of R_broadcast() generates at most $O(n^2)$ sending of m.

Reliable broadcast in the presence of Byzantine processes (BRB) In the Byzantine failure context, it is not possible to control the message r-deliveries of the Byzantine process. The definition becomes the following one.

- If a non-faulty process p_i r-delivers a message m from a non-faulty process p_j, m has previously been r-broadcast by p_j.
- Integrity. Assuming all the messages are different, no non-faulty process r-delivers twice the same message (no duplication).
- Termination-1. If a non-faulty process r-broadcasts a message m, it r-delivers m.
- Termination-2. If a non-faulty process r-delivers a message m, (whatever the sender of m) every non-faulty process r-delivers the message m.

Similarly to the case of crash failures, the following agreement property follows from the previous properties: the non-faulty processes delivers the same set of messages M, which contains (at least) all the messages they r-broadcast.

On Byzantine reliable broadcast algorithms Bracha presented in [15] an elegant signature-free algorithm, which implements the reliable broadcast abstraction in the presence of asynchrony and up to $t < n/3$ Byzantine processes. It is proved in [15, 16] that $t < n/3$ is an upper bound for the number of Byzantine processes, hence Bracha's algorithm is optimal from a t-resilience point of view. From an operational point of view, this algorithm is based on a *double echo* mechanism of the value broadcast by the sender process. For each application message, this algorithm requires three consecutive communication steps, and generates $(n-1)(2n+1)$ implementation messages.

It is natural that, as it is a fundamental communication abstraction, Byzantine reliable broadcast has been addressed by many authors. Here are a few recent results.

- The BRB algorithm presented in [50], implements the broadcast of an application message with only two communication steps (*single echo* mechanism), two message types, and $n^2 - 1$ protocol messages. The price to pay for this gain in efficiency is a weaker t-resilience, namely $t < n/5$. Hence, this algorithm and Bracha's algorithm differ in the trade-off t-resilience versus message/time efficiency.
- Scalable BRB is addressed in [39]. The issue is here not to pay the $O(n^2)$ message complexity cost. To this end, the authors use a non-trivial message-gossiping approach which allows them to design a sophisticated BRB algorithm satisfying fixed probability-dependent properties.
- BRB in *dynamic* systems is addressed [38]. Dynamic means that a process can enter and leave the system at any time. In their article the authors present an efficient BRB algorithm for such a context. This algorithm assumes that, at any time, the number of Byzantine processes present in the system is less than one third of total number of processes present in the system.
- An efficient algorithm for BRB with long inputs of ℓ bits using lower costs than ℓ single-bit instances is presented in [69]. This algorithm, which assumes $t < n/3$, achieves the best possible communication complexity of $\Theta(n\ell)$ input sizes. This article also presents an authenticated extension of the previous algorithm.

7 Further Readings

The reader interested in dissymetric progress conditions will consult [48, 51, 88]. A tutorial on the notion of universality in crash-prone asynchronous message-passing systems is presented in [78]. There are several textbooks entirely devoted to fault-tolerant distributed computing, e.g. [10, 17, 37, 44, 52, 55, 59, 74, 75, 77, 82, 87] to cite a few. Differently, the textbook [75] is entirely devoted to algorithms for failure-free asynchronous distributed systems.

In the today blockchain-dominated technology, the previous readings could help better understand what can be done and which are the assumptions needed to obtain provably reliable distributed software.

Acknowledgments

This work was partially supported by the French ANR project ByBLoS (ANR-20-CE25-0002-01) devoted the modular design of building blocks for large-scale trustless multi-users applications.

References

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

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

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

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

[5] Aguilera M.K. and Toueg S., A simple bi-valency proof that t-resilient consensus requires $t + 1$ rounds. *Information Processing Letters*, 71:155-158 (1999)

[6] Anderson J.H., Multi-writer composite registers. *Distributed Computing*, 7(4):175-195 (1994)

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

[8] Attiya H., Bar-Noy A., Dolev D., Peleg D., and Reischuk R., Renaming in an asynchronous environment. *Journal of the ACM*, 37(3):524-548 (1990)

[9] Attiya H. and Rajsbaum S., Indistinguishability. *Communications of the ACM*, ACM 63(5):90–99 (2020)

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

[11] Ben-Or M., Another advantage of free choice: completely asynchronous agreement protocols. *Proc. 2nd ACM Symposium on Principles of Distributed Computing (PODC'83)*, pp. 27-30 (1983)

[12] Ben-Or M., Kelmer B., and Rabin T., Asynchronous secure computations with optimal resilience. *Proc. 13th ACM Symposium on Principles of Distributed Computing (PODC'94)*, ACM Press, pp. 183-192 (1994)

[13] Borowsky E. and Gafni E., Immediate atomic snapshots and fast renaming. *Proc. 12th ACM Symposium on Principles of Distributed Computing (PODC'93)*, ACM Pres, pp. 41-51 (1993)

[14] Bouzid Z., Mostéfaoui, and Raynal M., Minimal synchrony for Byzantine consensus. *Proc. 34th ACM Symposium on Principles of Distributed Computing (PODC'15)*, ACM Press, pp. 461-470, (2015)

[15] Bracha G., Asynchronous Byzantine agreement protocols. *Information & Computation*, 75(2):130-143, (1987)

[16] Bracha G. and Toueg S., Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4):824-840 (1985)

[17] Cachin Ch., Guerraoui R., and Rodrigues L., *Reliable and secure distributed programming*, Springer, 367 pages, ISBN 978-3-642-15259-7 (2011)

[18] Castañeda A., Rajsbaum S., and Raynal M., The renaming problem in shared memory systems: an introduction. *Elsevier Computer Science Review*, 5:229-251 (2011)

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

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

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

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

[23] Correia M., Ferreira Neves N., and Veríssimo P., How to tolerate half less one Byzantine nodes in practical distributed systems. *Proc. 23rd Int'l Symposium on Reliable Distributed Systems (SRDS'04)*, IEEE Press, pp. 174-183 (2004)

[24] Correia M., Ferreira Neves N., and Veríssimo P., BFT-TO: intrusion tolerance with less replicas. *The Computer Journal*, 56(6):693-715 (2013)

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

[26] Dolev D., Lynch N. A., Pinter S. H., Stark E. W., and Weihl W. E., Reaching approximate agreement in the presence of failures. *Journal of the ACM*, 33(3):499-516 (1986)

[27] Dolev D., Reischuk, R. Strong H.R., Early stopping in Byzantine agreement. *Journal of the ACM*, 37(4):720-741 (1990)

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

[29] Dutta P., Guerraoui R., Levy R., and Vukolic M., Fast access to distributed atomic memory. *SIAM Journal of Computing*, 39(8):3752-3783 (2010)

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

[31] Ellen F., Fatourou P., Kosmas E., Milani A., and Travers C., Universal constructions that ensure disjoint-access parallelism and wait-freedom. *Distributed Computing*, 29:251-277 (2016)

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

[33] Fatourou P. and Kallimanis N.D., Highly-efficient wait-free synchronization. *Theory of Computing Systems*, 55:475-520 (2014)

[34] Fischer M.J. and Lynch N.A., A lower bound for the time to ensure interactive consistency. *Information Processing Letters*, 14:183-186 (1982)

[35] 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)

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

[37] Garg V.K., *Elements of Distributed Computing*. Wiley-Interscience, 423 pages (2002)

[38] Guerraoui G., Komatovic J., Kuznetsov P., Pignolet P.A., Seredinschi D.-A., and , Tonkikh A., Dynamic Byzantine reliable broadcast. *Proc. 24th Int'l Conference on Principles of Distributed Systems (OPODIS'20)*, Lipics Vol. 184, Article 23, 18 pages (2020)

[39] Guerraoui G., Kuznetsov P., Monti M., Pavlovic M., and Seredinschi D.-A., Scalable Byzantine reliable broadcast. *Proc. 33rd Int'l Symposium on Distributed Computing (DISC'19)*, LIPIcs Vol. 146, Article 22, 16 pages (2019)

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

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

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

[43] Herlihy M., Rajsbaum S., Raynal M., and Stainer J., From wait-free to arbitrary concurrent solo executions in colorless distributed computing. *Theoretical Computer Science*, 683:1-21 (2017)

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

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

[46] Imbs S., Rajsbaum S., Raynal M., and J. Stainer., Read/Write shared memory abstraction on top of an asynchronous Byzantine message-passing system. *Journal of Parallel and Distributed Computing*, 93-94:1-9 (2016)

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

[48] Imbs D. and Raynal M., A liveness condition for concurrent objects: x-wait-freedom. *Concurrency and Computation: Practice and experience*, 23:2154-2166 (2011)

[49] Imbs D. and Raynal M., Help when needed, but no more: efficient read/write partial snapshot. *Journal of Parallel and Distributed Computing*, 72(1):1-13 (2012)

[50] Imbs D. and Raynal M., Trading t-resilience for efficiency in asynchronous Byzantine reliable broadcast. *Parallel Processing Letters*, Vol. 26(4), 8 pages (2016)

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

[52] Kshemkalyani A.D. and Singhal M., *Distributed computing: principles, algorithms and systems*. Cambridge University Press, 736 pages (2008)

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

[54] Lamport L., Message-Id: <8705281923.AA09105@jumbo.dec.com>, Thu, 28 May 87 12:23:29

[55] Lamport L., *Specifying systems*. Addison-Wesley, Pearson Education, 364 pages (2003)

[56] Lamport L., Teaching concurrency. *ACM Sigact NEWS*, 40(1):58-62 (2009)

[57] Lamport L., Shostack R. and Pease M., The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3)-382-401 (1982)

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

[59] Lynch N.A., *Distributed algorithms*. Morgan Kaufmann Pub., San Francisco (CA), 872 pages, ISBN 1-55860-384-4 (1996)

[60] 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)

[61] Mostéfaoui A., Mourgaya E., and Raynal M., Asynchronous implementation of failure detectors. *Proc. Int'l IEEE Conference on Dependable Systems and Networks (DSN'03)*, IEEE Computer Society Press, pp. 351-360 (2003)

[62] Mostéfaoui A., Perrin M., and Raynal M., A simple object that spans the whole consensus hierarchy. *Parallel Processing Letters*, 28(2):1850006:1-1850006:9 (2018)

[63] Mostéfaoui A., Petrolia M., Raynal M., and Jard C., Atomic read/write memory in signature-free Byzantine asynchronous message-passing systems. In *Theory of Computing Systems*, Springer, 60(4):677-694 (2017)

[64] 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)

[65] Mostéfaoui A. and Raynal M., Intrusion-tolerant broadcast and agreement abstractions in the presence of Byzantine processes. *IEEE Transactions on Parallel and Distributed Systems*, 27(4):1085-1098 (2016)

[66] Mostéfaoui A. and Raynal M., Two-bit messages are sufficient to implement atomic read/write registers in crash-prone systems. *Proc. 35th ACM Symposium on Principles of Distributed Computing (PODC'16)*, ACM Press, pp. 381-390 (2016)

[67] Mostéfaoui A. and Raynal M., Signature-free asynchronous Byzantine systems: from multivalued to binary consensus with $t < n/3$, $O(n^2)$ messages, and constant time. *Acta Informatica*, 54:501âĂŞ520 (2017)

[68] Mostéfaoui A., Raynal M., and Roy M., Time-efficient read/write register in crash-prone asynchronous message-passing systems. *Springer Computing*, 101:3–17 (2018)

[69] Nayak K., Ren L., Shi E., Vaidya N.H., Xiang Z., Improved extension protocols for Byzantine broadcast and agreement. *Proc. 34rd Int'l Symposium on Distributed Computing (DISC'20)*, LIPIcs Vol. 179, Article 28, 16 pages (2020)

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

[71] Perrin M., Mostéfaoui A., and Bonin G., Extending the wait-free hierarchy to multi-threaded systems. *Proc. 39th ACM Symposium on Principles of Distributed Computing (PODC'20)*, ACM Press, pp. 21–30 (2020)

[72] Rabin M., Randomized Byzantine generals. *Proc. 24th IEEE Symposium on Foundations of Computer Science (FOCS'83)*, IEEE Computer Society Press, pp. 116-124 (1983)

[73] Rajsbaum S. and Raynal M., Mastering concurrent computing through sequential thinking: A half-century evolution. *Communications of the ACM*, Vol. 63(1):78-87 (2020)

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

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

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

[77] Raynal M., *Fault-tolerant message-passing distributed systems: an algorithmic approach.* Springer, 550 pages, ISBN: 978-3-319-94140-0 (2018)

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

[79] Raynal M., An Informal visit to the wonderful land of consensus numbers and beyond. *Bulletin of the European Association of Theoretical Computer Science (EATCS)*, 129:168-192 (2019)

[80] Raynal M., and Stainer J., Round-based synchrony weakened by message adversaries vs asynchrony enriched with failure detectors. *32th ACM Symposium on Principles of Distributed Computing (PODC'13)*, ACM Press, pp. 166-175 (2013)

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

[82] Santoro N., *Design and analyis of distributed algorithms*, Wiley-Interscience, 589 pages, ISBN 0-471-71997-8 (2007)

[83] Santos Veronese G., Correia M., Bessani A., Lung C.L., and Veríssimo P., Efficient Byzantine fault-tolerance. *IEEE Transactions on Computers* 62(1):16-30 (2013)

[84] Schneider F.B., Implementing fault-tolerant services using the state machine approach. *ACM Computing Surveys*, 22(4):299-319 (1990)

[85] Schneider F.B., What good are models, and what models are good? *Distributed Systems (2nd Edition)*. Adddison-Wesley/ACM Press, pp. 17-26 (1993)

[86] Shapiro M., Preguiça N.M., Baquero C., and Zawirski M., Conflict-free replicated data yypes. *Proc. 13th Int'l Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'11)*, Springer LNCS 6976, pp. 386-400 (2011)

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

[88] Taubenfeld G., The computational structure of progress conditions and shared objects. *Distributed Computing*, 33(2):103-123 (2020)

[89] Toueg S., Randomized Byzantine agreement. *Proc. 3rd Annual ACM Symposium on Principles of Distributed Computing (PODC'84)*, ACM Press, pp. 163-178 (1984)

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

[91] Turpin R. and Coan B.A., Extending binary Byzantine agreement to multivalued Byzantine agreement. *Information Processing Letters*, 18:73-76 (1984)

[92] Parallel computing. https://en.wikipedia.org/wiki/Parallel_computing