

# Signature-Free Communication and Agreement in the Presence of Byzantine Processes

Michel Raynal\*

Institut Universitaire de France, Paris, France; and  
IRISA, Université de Rennes, Rennes, France

---

## Abstract

Communication and agreement are fundamental abstractions in any distributed system. (If the computing entities do not need to communicate or agree in one way or another, the system is not a distributed system!) This tutorial was devoted to the design of such abstractions built on top of signature-free asynchronous distributed systems prone to Byzantine process failures. It is made up of three parts, each devoted to an abstraction and algorithms that implement it.

**1998 ACM Subject Classification** C.2.4 [Computer-Communication Network] Distributed Systems – distributed applications, network operating systems, D.4.5 [Operating Systems] Reliability – fault-tolerance, F.1.1 [Computation by Abstract Devices] Models of Computation, Computability theory

**Keywords and phrases** Asynchronous system, Atomic read/write register, Byzantine process, Consensus, Distributed algorithm, Distributed computability, Fault-tolerance, No-duplicity broadcast, Process crash, Reliable broadcast

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2015.1

**Category** Tutorial

## 1 Introduction

### 1.1 Aim of the tutorial

This tutorial was motivated by the following observations:

- The world is distributed and more and more applications are distributed.
- Asynchronous message-passing systems are more and more pervasive.
- In one way or another, computing entities have to communicate and agree.
- The assumption “no computing entity has a bad behavior” is no longer reasonable.

Its aim was consequently to present basic communication and agreement abstractions which can cope with bad (intentional or not) behavior of a subset of the computing entities.

**On the content of this article.** This companion text presents the definition of the abstractions addressed in the tutorial. The page limitation does not allow to present the algorithms that implement them. The reader can find them in the corresponding articles or technical reports.

---

\* This work has been partially supported by the French ANR project DISPLEXITY devoted to computability and complexity in distributed computing, the Franco-German ANR project DISCMAT devoted to connections between mathematics and distributed computing.



© Michel Raynal;  
licensed under Creative Commons License CC-BY

19th International Conference on Principles of Distributed Systems (OPODIS 2015).

Editors: Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Gradinariu; Article No. 1; pp. 1:1–1:10

Leibniz International Proceedings in Informatics



LIPIC Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



## 1.2 Underlying computation model

**Processes.** The computing model is composed of  $n$  processes (computing entities), denoted  $p_1, \dots, p_n$ , which are sequential and asynchronous (each process progresses at its own speed which is arbitrary and never known by the other processes).

**Communication medium.** Any pair of processes is connected by a bidirectional communication channel which allows them to send and receive messages. The underlying network is consequently fully connected, allowing a receiver to know which process sent the message it receives. It is assumed to be reliable: there is neither loss, creation, duplication, nor alteration of messages.

**Process failure model.** A process that executes correctly its local algorithm is said to be *correct*. A process that does not execute correctly its local algorithm is said to be *faulty*. We consider here the severe failure model, where a faulty process can behave arbitrarily, which is called *Byzantine* behavior [21, 30].

A Byzantine process can crash (premature halt), execute arbitrary code, omit to send or receive messages, send erroneous messages, decide not to send a message when it is assumed to do it, send different values to different processes when it is assumed to send them the same value, etc. Byzantine processes can also collude to foil the correct processes.

Let us observe that a correct process can never know if another process is correct or Byzantine. Moreover, a Byzantine process can behave as if it was correct during arbitrary long periods separated by faulty behavior.

**On signatures.** Digital signatures (based on public key cryptosystems) can be used to restrain the bad behavior of Byzantine processes. As an example, a Byzantine process which must forward a message signed by a correct process can only forward it or not forward it. It cannot corrupt its content.

Signatures have a computation cost, and require the adversary (here the set of Byzantine processes) to be computationally limited. Moreover, when data are not confidential, signatures are not always necessary to cope with malicious behavior [29, 33]. Hence, the tutorial considered signature-free systems. This allows the adversary to have an unbounded computational power.

**Constraint and notation.** Let  $t$  be the upper bound on the number of Byzantine processes. It is assumed in the following that  $t < n/3$ . This requirement is necessary to solve the problems in which we are interested here. The corresponding computing system is denoted  $\mathcal{BZ\_AS}_{n,t}[t < n/3]$ .

## 2 Reliable Broadcast Abstractions

The aim of a broadcast abstraction is to allow a correct process to send a value to all correct processes, with some provable delivery guarantees. The tutorial considered two of them.

### 2.1 No-duplication broadcast

The no-duplication broadcast (ND-broadcast) was introduced by S. Toueg in [34]. It provides the processes with the operations `ND_broadcast()` and `ND_deliver()`, which allows any correct

process to broadcast a message (we say “ND-broadcast”) and to deliver messages (we say “ND-deliver”) while guaranteeing the following properties:

- ND-Validity: If a correct process ND-delivers a message from a correct  $p_i$ , then  $p_i$  invoked `ND_broadcast()`.
- ND-no-duplication: No two correct processes ND-deliver distinct messages from the same (correct or Byzantine) process  $p_i$ .
- ND-Termination-1: If the sender is correct, all the correct processes eventually ND-deliver its message.

This specification can easily be extended to the case where a process needs to ND-broadcast several messages. In this case a tag must be associated with each message (e.g., a pair  $\langle$  sender id, sequence number  $\rangle$ ) and a correct process can ND-deliver only once a message with a given tag (e.g., [7]).

The ND-broadcast abstraction ensures that no two correct processes ND-deliver different messages from the same ND-broadcaster. A message ND-broadcast by any correct process is ND-delivered by each correct process, and no fake message is ND-delivered from a correct process.

Considering an ND-broadcast instance whose sender is a Byzantine process  $p_k$ , If a correct process  $p_i$  ND-delivers a message  $m_1$  while another correct process  $p_j$  ND-delivers a message  $m_2$ , we necessarily have  $m_1 = m_2$ . According to the previous specification, it is nevertheless possible that  $p_i$  ND-delivers  $m_1$  while  $p_j$  does not deliver a message from  $p_k$ . As shown in the tutorial this communication abstraction can be implemented in  $\mathcal{BZ\_AS}_{n,t}[t < n/3]$ .

Let us notice that, when process ND-broadcast several (tagged) messages, ND-broadcast does not prevent two correct processes from ND-delivering different sets of messages. These sets may differ in the messages from Byzantine processes: a message can appear in one set and not in another set.

## 2.2 Reliable broadcast

The reliable broadcast abstraction was introduced in several articles. We consider here a definition close to the given by Bracha [6]. Its associated operations are denoted `RB_broadcast()` and `RB_deliver()`.

This communication abstraction can be seen as ND-broadcast enriched with an additional termination property, stating that all correct processes RB-deliver the same set of messages, this set including at least all the messages RB-broadcast by the correct processes. More formally, RB-broadcast is defined by the following properties:

- RB-Validity: If a correct process RB-delivers a message from a correct process  $p_i$ , then  $p_i$  invoked `RB_broadcast()`.
- RB-no-duplication: No two correct processes RB-deliver distinct messages from the same (correct or Byzantine) process  $p_i$ .
- RB-Termination-1: If the sender is correct, all correct processes eventually RB-deliver its message.
- RB-Termination-2: If a correct process RB-delivers a message  $m$  from  $p_i$  (possibly Byzantine), all the correct processes eventually RB-deliver  $m$  from  $p_i$ <sup>1</sup>.

---

<sup>1</sup> A similar modular presentation with a separation of Termination-1 and Termination-2, which clarifies the relation between ND- and RB-broadcasts appeared in [8], where the communication abstractions are called consistent broadcast and reliable broadcast, respectively.

This broadcast abstraction can be built in  $\mathcal{BZ\_AS}_{n,t}[t < n/3]$ , on top of the ND-broadcast abstraction. The algorithm described in [6] generates  $O(n^2)$  implementation messages, and, assuming each message takes one time unit, the RB-delivery of a message requires three time units. Moreover, three types of causally-related implementation messages are used in Bracha's algorithm [6]. Recently, a new RB-broadcast algorithm has been proposed [18]. This implementation requires  $O(n^2)$  implementation messages (as [6]), but has a smaller time complexity, namely 2 instead of 3.

### **3 Read/write Register Abstraction**

#### **3.1 Read/Write Register**

A read/write register is the most basic object encountered in computing science. It provides the processes two operations, denoted `read()` and `write()`, which allow the invoking process to obtain the value of the register and assign a new value to the register, respectively.

**On the different types of registers.** In the presence of concurrency, several processes may concurrently access a register. In such a context, according to the requirement on the values returned by the read invocations, several types of registers can be defined [19, 20], namely safe, regular, and atomic. From a computability point of view, they have the same power in the presence of process crashes (see the textbooks [2, 23, 32]). There is nevertheless a cost to go from safe registers to regular registers, and from regular registers to atomic registers.

**Atomic register.** We consider here atomic registers. A register is *atomic* [19] (or linearizable [16]) if its read and write operations satisfy the following properties:

- The execution of each operation appears as if it has been executed at a single point of the time line between its start event and its end event,
- No two operations appear at the same point of the time line, and
- Each read returns the last value written before it in the sequence.

#### **3.2 An Implementation in the process crash failure model**

A simple and elegant algorithm implementing an atomic register in the asynchronous message-passing model where up to  $t < n/2$  may commit crash failure was introduced in [1]. This paper shows also that  $t < n/2$  is a necessary requirement for such an implementation. This algorithm is based on the following principles and mechanisms:

- An increasing sequence number is associated with each written value,
- Each process manages a local copy of the register,
- Using request and acknowledgment messages, each write operation updates a majority of local copies,
- Similarly, using request and acknowledgment messages, each read operation obtains  $\langle \text{value}, \text{seq. number} \rangle$  pairs from a majority of processes, and returns the value whose sequence number  $sn$  is the greatest. Moreover, to ensure atomicity, before returning, the read operation must ensure that a majority of local copies have a value whose sequence number is  $\geq sn$ .

#### **3.3 Atomic registers in the Byzantine failure model**

Parts of this section are from [26].

**Single-writer multi-reader register.** As it is not possible to constrain the behavior of a Byzantine process, such a process can corrupt any register it can access. Hence, implementing atomic registers in the presence of Byzantine processes is meaningful only if we consider single-writer multi-reader (SWMR) registers, i.e., registers that can be written by a single process, but read by any process. In this way a Byzantine process can only corrupt the registers for which it is the only writer.

Hence, we consider here the construction of an array of SWMR registers  $REG[1..n]$ , such that  $REG[i]$  can be written only by  $p_i$ .

**Read and write by a Byzantine process.** As a Byzantine process can behave arbitrarily, there is no requirement on the value it returns from a read invocation.

As far as the write operation is concerned, the situation is more complicated. A Byzantine process  $p_k$  can invoke  $REG[k].write()$  as if it was correct. It can also try to modify (by generating appropriate implementation messages) the content of  $REG[k]$  without invoking  $REG[k].write()$ . If it succeeds, the corresponding modification of  $REG[k]$  is considered as if it was produced by an invocation of  $REG[k].write()$ . This is because, these two cases cannot be distinguished by the correct processes. Let us also notice that it is not possible to prevent a value written by a Byzantine process from being a value that has nothing to do with the problem to be solved (*fake* value).

**Preliminary definitions.** The specification of an array  $REG[1..n]$  of atomic SWMR registers in a Byzantine failure context is based on the following definitions:

- An *abstract* sequence  $H_i$  is associated with each register (intuitively,  $H_i$  represents the sequence of values written by  $p_i$  in its register  $REG[i]$ ).
- If  $p_i$  is correct, let  $read[i, j, x]$  denote the execution by  $p_i$  of  $REG[j].read()$  returning the value of  $H_j[x]$ .
- Let  $write[i, x]$  denote the  $x$ th modification of  $REG[i]$  (necessarily by  $p_i$ ):
  - If  $p_i$  is correct,  $write[i, x]$  is the  $x$ th execution of  $REG[i].write()$  by  $p_i$ .
  - If  $p_i$  is Byzantine,  $write[i, x]$  is the  $x$ th modification of  $REG[i]$  by  $p_i$  (not necessarily due to an invocation of  $REG[i].write()$ ).

**Specification.** Each register of the array  $REG[1..n]$  is defined by the following set of properties:

- Termination. If  $p_i$  is a correct process, all its invocations of  $REG[i].write()$  terminate, and for any  $j$ , all its invocations of  $REG[j].read()$  terminate.
- Atomicity. Let  $p_i$  and  $p_j$  be two correct processes, and  $p_k$  a correct or Byzantine process.
  - Read followed by write: ( $read[i, k, x]$  terminates before  $write[k, y]$  starts)  $\Rightarrow (x < y)$ .
  - Write followed by read: ( $write[j, x]$  terminates before  $read[i, j, y]$  starts)  $\Rightarrow (x \leq y)$ .
  - No read inversion: ( $read[i, k, x]$  terminates before  $read[j, k, y]$  starts)  $\Rightarrow (x \leq y)$ .

The first rule states that a process cannot read from the future. The second rule states that a read cannot obtain an overwritten value. The last rule states that, given any register  $REG[k]$ , sequential read operations concurrent with one or several write operations must respect the sequential order on these writes.

It is easy to show that, from these rules, each register behaves atomically [19]<sup>2</sup>, i.e., as defined in Section 3.1.

---

<sup>2</sup> Or is linearizable according to the terminology of [16].

**From message-passing to read/write registers.** Algorithms implementing atomic registers in  $\mathcal{BZ\_AS}_{n,t}[t < n/3]$  are described in [17, 26]. As shown in [17],  $t < n/3$  is a necessary requirement for such an implementation.

Such constructions allow us to execute algorithms designed to run on an SWMR shared memory where at most  $t < n/3$  processes may be Byzantine, on top of an asynchronous message-passing system. This is important because designing a Byzantine-tolerant algorithm for a given problem is usually easier in the shared memory context than in the message-passing context. Examples of such algorithms are described in [17].

## 4 Agreement Abstraction (Consensus)

### 4.1 The consensus problem

**Definition.** Consensus is one of the most (maybe the most) important agreement problem of fault-tolerant distributed computing. Its informal statement is extraordinary simple: it requires that all correct processes agree of the same value.

More precisely, assuming that each process proposes a value, the consensus abstraction in a Byzantine failure context, is defined by the following properties:

- C-Termination: Every correct process eventually decides a value.
- C-One-shot: A correct process decides at most once.
- C-Agreement: No two correct processes decide different values.
- C-Validity: If all correct processes propose the same value  $v$ , then  $v$  is decided.

If only two values can be proposed, consensus is binary. Otherwise it is multivalued. In the following we consider binary consensus. This is not a problem, as there exist algorithms, for asynchronous systems, which solve multivalued consensus on top of binary consensus (e.g., [8, 28, 32] to cite a few).

**Impossibility in asynchronous systems.** Despite its very simple statement, there is no deterministic algorithm that solves the consensus problem in the presence of asynchrony and failures:

- as soon as  $n \geq 2$ ,
- whatever the communication medium (read/write shared memory or message-passing),
- even if only a single process may fail,
- even if the process failure model is the less severe, namely process crash,
- even if the processes have to agree on a single bit.

This impossibility is a foundation result of fault-tolerant distributed computing. It states a limit of what can be computed in the presence of asynchrony and failures, in the context of read/write communication, or message-passing communication. It has first been established in the context of asynchronous message-passing systems by Fischer, Lynch and Paterson, hence its name “FLP impossibility” result in 1985 [13]. It has then been extended to read/write shared memory systems in 1987 [22].

The intuition that underlies this impossibility lies in the fact that, due to asynchrony, a process is unable to know if another process has crashed or is only very slow (or its incident channels are very slow).

**How to circumvent the consensus impossibility.** Several approaches have been proposed to circumvent the previous impossibility. We cite here three of them.

- Enrich the system with information on failures. This is the failure detector approach introduced in [11]. This approach allowed to discover the weakest information on failures needed to solve some problems (otherwise impossible to solve). The weakest failure detector to solve consensus in the crash failure model is called  $\Omega$  [12]. It states that all correct processes must eventually agree on the same leader, which must be one of them.
- Restrict the set of input vectors, where an input vector is an  $n$ -size vector, each of its entries containing the value proposed by the corresponding process [27]. The problem consists then in defining the greatest sets of input vectors (each such set is called a condition), such that, given a condition, consensus can be solved if and only if the input vector belongs to the condition.

Interestingly, a strong connection relating conditions (hence, the consensus problem) with error-correcting codes has been established in [14].

- Enrich the system with randomization. In this case, a process can draw random numbers to face the uncertainty created by the net effect of asynchrony and failures. In this case, the algorithms are no longer deterministic. This approach was introduced to solve consensus in 1983 by M. Rabin [31] and M. Ben-Or [3].

As far as consensus is concerned, the C-Termination property becomes “Each correct process decides with probability 1”. In round-based algorithms (as are consensus algorithms), this property translates as follows

$$\lim_{r \rightarrow +\infty} \text{Proba}[p_i \text{ decides by round } r] = 1.$$

This is the approach we consider in the following.

## 4.2 Related works

This section borrows parts (including the table) from [25]. Some of the first randomized consensus algorithms (e.g. [3, 5]) use *local coins* (the values returned to a process by a local coin is not related to the values returned by the local coins of the other processes). As a consequence, they have an expected number of rounds which is exponential (unless  $t = O(\sqrt{n})$ ). As randomized algorithms based on a common coin can have an expected number of rounds which is constant, this paper focuses only on such algorithms.

**Common coin.** As defined in [9], a *common coin* is a global entity that delivers the same sequence of random bits to each process, each value with probability 1/2.

To obtain a random bit, a process invokes the operation `random()`, whose  $r$ th invocation by any process returns it the bit  $b_r$ . Hence, all correct processes obtain the same sequence of random bits  $b_1, b_2, \dots, b_r$ , etc.

Moreover, it is not possible for Byzantine process to obtain bits in advance, namely, a Byzantine process can obtain the value of  $b_r$  only when at least one correct process has started accessing  $b_r$ . It follows that a common coin is a strongly synchronized coin (see [9] for the implementation of a common coin).

Randomized asynchronous Byzantine algorithms using a common coin are listed in Table 1 (at the last line,  $\ell$  denotes the length of an RSA signature). All these algorithms, which address binary consensus, are based on asynchronous rounds, and, in each of them, every message carries the number of the round in which it is sent. Hence, when comparing their message size, we do not consider round numbers. We have the following.



■ **Table 1** Cost and constraint of different Byzantine binary consensus algorithms.

Protocol	$n >$	sign.	msgs/round	bits/msg	steps/rd
Rabin [31]	$10t$	yes	$O(n^2)$	$O(1)$	2
Berman Garay [4]	$5t$	no	$O(n^2)$	$O(1)$	2
Friedman Mostéfaoui Raynal [15]	$5t$	no	$O(n^2)$	$O(1)$	1
Bracha [6]	$3t$	no	$O(n^3)$	$O(\log(n))$	9
Srikanth Toueg [33]	$3t$	no	$O(n^3)$	$O(\log(n))$	5
Toueg [34]	$3t$	yes	$O(n^3)$	$O(n)$	3
Canetti Rabin [10]	$3t$	no	$O(n^2)$	$\text{poly}(n)$	9
Cachin Kursawe Shoup [9]	$3t$	yes	$O(n^2)$	$O(\ell)$	2

- The first algorithm is such that  $n < 10t$ , has an  $O(n^2)$  message complexity, and requires signatures.
- The algorithms of the two next lines are such that  $t < n/5$ , and their message complexity is  $O(n^2)$ . These algorithms are simple, signature-free, and use one or two communication steps per round, but none of them is optimal with respect to  $t$ -resilience.
- The algorithms of the next three lines are optimal with respect to  $t$ , but have an  $O(n^3)$  message complexity. Moreover, [34] uses signed messages (to prevent message falsification by Byzantine processes), while [6] does not use a common coin, and may consequently execute an exponential number of rounds. Due to their message complexity, these algorithms are costly.
- As far as the last two lines of the table are concerned, we have the following. Both are optimal with respect to the resilience parameter  $t$ , and the number of message per round  $O(n^2)$ , and use signed messages. The algorithm proposed in [10], although polynomial, has a huge bit complexity. The algorithm presented in [9] has two communication steps per round.

### 4.3 An optimal algorithm

Contrarily to what could be “falsely deduced” from a quick reading at the randomized consensus algorithms cited in Table 1, the formula

$$[\text{quadratic message complexity}] \Rightarrow [(\text{use of signatures}) \vee (t < n/5)]$$

is false.

There is an algorithm (first introduced in [24], and then improved and generalized in [25] to allow the use of a weak common coin) that has the following set of properties:

- The algorithm 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.
- Byzantine processes may re-order messages sent to correct processes.
- The algorithm uses a weak 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).



## 5 Conclusion

The aim of this paper was to provide the reader with the main ideas, and concepts presented in the tutorial given at OPODIS 2015 (which additionally includes corresponding algorithms). The interested reader can obtain a deeper knowledge of the topic by reading papers listed in the bibliography.

**Acknowledgments.** I want to thank Achour Mostéfaoui with whom I worked a lot on Byzantine failures, and Christian Cachin for a careful reading of this paper and constructive comments, which helped improve its presentation.

---

### References

- 1 Attiya H., Bar-Noy A. and Dolev D., Sharing memory robustly in message passing systems. *Journal of the ACM*, 42(1):121–132 (1995)
- 2 Attiya H. and Welch J., *Distributed computing: fundamentals, simulations and advanced topics*, Wiley-Interscience, 414 pages (2004)
- 3 Ben-Or M., Another advantage of free choice: completely asynchronous agreement protocols. *Proc. 2nd ACM Symposium on Principles of Distributed Computing(PODC'83)*, ACM Press, pp. 27–30 (1983)
- 4 Berman P. and Garay J.A., Randomized distributed agreement revisited. *Proc. 33rd Annual Int'l Symposium on Fault-Tolerant Computing (FTCS'93)*, IEEE Computer Press, pp. 412–419 (1993)
- 5 Bracha G., An asynchronous  $(n - 1)/3$ -resilient consensus protocol. *Proc. 3rd Annual ACM Symposium on Principles of Distributed Computing (PODC'84)*, ACM Press, pp. 154–162 (1984)
- 6 Bracha G., Asynchronous Byzantine agreement protocols. *Information & Computation*, 75(2):130–143 (1987)
- 7 Cachin Ch., State machine replication with Byzantine failures. In *Replication: Theory and Practice*, Springer LNCS 5959, pp. 169–174 (2010)
- 8 Cachin Ch., Kursawe K., Peztold F., and Shoup V., Secure and efficient asynchronous broadcast protocols. *Proc. 21st Annual International Cryptology Conference (CRYPTO'01)*, Springer LNCS 2139, pp. 524–543 (2001)
- 9 Cachin Ch., Kursawe K., and Shoup V., Random oracles in Constantinople: practical asynchronous Byzantine agreement using cryptography. *Journal of Cryptology*, 18(3):219–246 (2005, first version: PODC 2000)
- 10 Canetti R., and Rabin T., Fast asynchronous Byzantine agreement with optimal resilience, *Proc. 25th Annual ACM Symposium on Theory of Computing (STOC'93)*, ACM Press, pp. 42–51 (1993)
- 11 Chandra T. and Toueg S., Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267 (1996)
- 12 Chandra T., Hadzilacos V., and Toueg S., The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722 (1996)
- 13 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)
- 14 Friedman R., Mostéfaoui A., Rajsbaum S., and Raynal M., Distributed agreement problems and their connection with error-correcting codes. *IEEE Transactions on Computers*, 56(7):865–875 (2007)

- 15 Friedman R., Mostéfaoui A., and Raynal M., Simple and efficient oracle-based consensus protocols for asynchronous Byzantine systems. *IEEE Transactions on Dependable and Secure Computing*, 2(1):46–56 (2005)
- 16 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)
- 17 Imbs D., Rajsbaum S., Raynal M., and Stainer J., Reliable shared memory abstractions on top of asynchronous Byzantine message-passing systems. *Proc. 21th Int'l Colloquium on Structural Information and Communication Complexity (SIROCCO'14)*, Springer LNCS 8576, pp. 37–53 (2014)
- 18 Imbs D. and Raynal M., Simple and efficient reliable broadcast in the presence of Byzantine processes. <http://arxiv.org/abs/1510.06882> (2015), submitted to publication.
- 19 Lamport L., On interprocess communication, Part I: basic formalism. *Distributed Computing*, 1(2):77–85 (1986)
- 20 Lamport L., On Interprocess Communication, Part II: Algorithms. *Distributed Computing*, 1(2):86–101 (1986)
- 21 Lamport L., Shostack R., and Pease M., The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401 (1982)
- 22 Loui M.C., and Abu-Amara H.H., Memory Requirements for Agreement Among Unreliable Asynchronous Processes. *Par. and Distributed Computing: vol. 4 of Advances in Comp. Research*, JAI Press, 4:163–183 (1987)
- 23 Lynch N.A., *Distributed algorithms*. Morgan Kaufmann Pub., San Francisco (CA), 872 pages (1996)
- 24 Mostéfaoui A., Moumen H., and Raynal M., Signature-free asynchronous Byzantine consensus with  $t < n/3$  and  $O(n^2)$  messages. *Proc. 33th ACM Symposium on Principles of Distributed Computing (PODC'14)*, ACM Press, pp. 2–9 (2014)
- 25 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)
- 26 Mostéfaoui A., Petrolia M., Raynal M., and Jard Cl., Atomic read/write memory in Signature-free Byzantine asynchronous message-passing systems. Tech Report 2028, IRISA, University of Rennes, France (2015), <https://hal.inria.fr/hal-01238765>.
- 27 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)
- 28 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. *Proc. 22nd Int'l Colloquium on Structural Information and Communication Complexity (SIROCCO'15)*, Springer LNCS 9439, pp. 194–208 (2015)
- 29 Mostéfaoui A. and Raynal M., Communication and agreement abstractions in the presence of Byzantine processes. To appear in *IEEE Transactions on Parallel and Distributed Systems* (2016)
- 30 Pease M., R. Shostak R., and Lamport L., Reaching agreement in the presence of faults. *Journal of the ACM*, 27:228–234 (1980)
- 31 Rabin M., Randomized Byzantine generals. *Proc. 24th IEEE Symposium on Foundations of Computer Science (FOCS'83)*, IEEE Press, pp. 116–124 (1983)
- 32 Raynal M., *Concurrent programming: algorithms, principles, and foundations*. Springer, 530 pages (2013) (ISBN 978-3-642-32026-2)
- 33 Srikanth T.K. and Toueg S., Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2:80–94 (1987)
- 34 Toueg S., Randomized Byzantine agreement. *Proc. 3rd Annual ACM Symposium on Principles of Distributed Computing (PODC'84)*, pp. 163–178 (1984)