The Distributed Computing Column

Seth Gilbert
National University of Singapore
seth.gilbert@comp.nus.edu.sg

This month, in the Distributed Computing Column, Michel Raynal revisits two classical problems: mutual exclusion and consensus. Both of these problems are central to distributed computing. Many date the birth of distributed computing, as a field, to Dijkstra's first paper on mutual exclusion in 1965, and it remains an area of active research today (see, e.g., new breakthroughs on recoverable mutual exclusion this year). Similarly, ever since the consensus problem was first formally defined in a 1980 paper by Pease, Shostak, and Lamport, it has been recognized as lying at the heart of distributed computing. Recent innovation in blockchains (which rely on consensus to order blocks) only reinforces the continued relevance of consenus today. Yet research on these two central problems has followed somewhat different paths, focusing on different models and different aspects of concurrency and fault-tolerance. In this short note, Michel Raynal provides a historical overview of their development, and argues that they are really "two sides of the same coin."

The Distributed Computing Column is particularly interested in contributions that propose interesting new directions and summarize important open problems in areas of interest. If you would like to write such a column, please contact me.

Mutual Exclusion vs Consensus: Both Sides of the Same Coin?

Michel Raynal
IRISA, CNRS, Inria, Univ Rennes, 35042 Rennes, France

Abstract

This short note shows that consensus is to logical objects what mutual exclusion (mutex) is to physical objects. Namely, both allow processes to cooperate in a consistent way through objects operations of which must be executed sequentially (i.e., objects defined by a sequential specification).

Keywords: Agreement, Asynchrony, Concurrency, Consensus, Liveness, Logical (immaterial) object, Mutual exclusion, Physical object, Safety, Sequential specification, Total order.

1 Concurrent Computing

While a sequential process describes the behavior of a given state machine [42], *concurrent computing* is about the study of asynchronous sequential processes that execute concurrently (i.e., possibly at the same time) but not independently from each other[1]. Asynchronous means that each process proceeds to its own speed, which can vary with time and remains always unknown to the other processes. The code executed by the processes can be specific or not to each process.

Considering a set of sequential processes, the concept of *concurrent processes* (multi-process program) captures the fact that the individual behavior of each sequential process must be controlled so that the global behavior of the set of processes remains consistent (which can be captured by predicates and invariants, e.g., [21, 22, 30, 31]). These fundamental notions have been introduced by E.W. Dijkstra in the early sixties [17, 18, 19, 20]. (The interested reader will find more historical and scientific developments in [2, 6, 32, 36, 39, 45, 53, 55, 57, 58].)

[1] At the very beginning, the corresponding underlying multiprocessor machine was simulated on a single mono-processor enriched with peripheral devices. Then it was a real physical multiprocessor. Today it is provided by what is sometimes called an *Internet machine* covering the world.

2 Parallel Computing vs Distributed Computing

Parallel computing Parallel computing is a natural extension of sequential computing in the sense that the aim of parallel computing is to detect and exploit *data independence* to obtain efficient programs: once identified, independent sets of data can be processed independently from each other on a multiprocessor. It is nevertheless important to notice that, while independent data can be processed in parallel, any parallel program could be executed on a single processor with an appropriate scheduler (the corresponding sequential execution could be of course highly inefficient!).

Distributed computing The nature of distributed computing is totally different. Namely, distributed computing is characterized by the fact that there is a set of predefined (and physically distributed) computing entities (processes) that are *imposed* to the programmers and these entities need to *cooperate to a common goal*. Moreover the behavior of the underlying infrastructure (also called environment) on which the distributed application is executed is not under the control of the programmers who have to consider it as an *hidden input*. Asynchrony and failures are the most frequent phenomenons produced by the environment that create a "context uncertainty" distributed computing has to cope with. In short, distributed computing is characterized by the fact that, in any distributed run, *the run itself is one of its entries* [54].

A duality To summarize, parallel computing is the exploitation of the independence of input data to obtain efficient algorithms (programs), while the aim of distributed computing is to allow predefined computing entities to cooperate to a common goal in a consistent way.

3 Mutex vs Consensus: Preliminary Remark

Both problems were originally addressed in different system models, namely *asynchronous systems with no failures* for mutex [18] (the only "adversary" was asynchrony), and *synchronous systems with Byzantine failures* for consensus [48] (the only "adversary" was Byzantine failures).

They were then extended to more general system models including both asynchrony and process failures, namely asynchronous systems with process crashes failures for mutex and asynchronous systems with process crashes/Byzantine failures for consensus. As we will see, this required to add computability power to the underlying system model for these problems can be solved.

4 1965: Mutual Exclusion

The very first objects that were shared by concurrent processes were *physical* objects (resources) such as discs, tapes, and shared memory. Mutual exclusion was then introduced to make their accesses by the processes consistent (what does happen if several processes simultaneously access such a physical object?). The problem and its answer were introduced by E.W. Dijkstra who proposed the notion of a *critical section*, namely a part of code that can be accessed by a single process at a time [18].

The mutex object To ensure this property, E.W. Dijkstra introduced a new concurrency-related object that we call here *mutex* (shortcut for mutual exclusion). This synchronization object provides processes with two operations denoted acquire() and release() that allow to bracket the critical section code as described by the following pattern:

acquire(); *critical section*; release().

As any computing object, the mutex object is specified with a set of properties that describe all its correct behaviors, namely:

- Mutual exclusion (safety). At most one process at a time executes the critical section.
- Starvation freedom (liveness). Any invocation of acquire() terminates (and consequently the invoking process eventually enters the critical section)[2]. (Let us observe that fact that any invocation of acquire() must terminate implies that any invocation of release() must also terminate.)

Encapsulation and sequential execution The two operations acquire() and release() are not visible at the application level. At this level a process invokes higher level operations, e.g., op() such that

operation op() **is** acquire(); *critical section*; release() **end operation**.

In some cases the object protected by a mutex object, say `mtr`, provides processes with several operations. This is for example the case of two resources $R1$ and $R2$ that, due to energy consumption, cannot be used at the very same time. In this case we have a single mutex object and two operations

operation opR1() **is** `mtr`.acquire(); *access* $R1$; `mtr`.release() **end operation**,

[2]When he introduced mutual exclusion, Dijkstra considered a weaker liveness property, named deadlock-freedom: If one or several processes invoke acquire(), at least one of them will enter the critical section.

and
operation opR2() **is** `mtr`.acquire(); *access R*2; `mtr`.release() **end operation**.

This approach has also naturally been used for data objects (for example a stack where opR1 is push() and opR2 is pop()).

It is easy to see that mutual exclusion allows processes to execute sequentially (we also say linearize [29]) predefined parts of code concerning their cooperation. So, mutual exclusion allows the processes to build a total order on the execution of the critical section codes protected by the same mutex object. From a historical point of view, mutex can be considered as the first distributed computing problem: it allows a predefined set of processes to cooperate to a common goal, namely preserve the consistency of an object in the presence of concurrency. A rigorous exposition of the mutex theory is presented in [38].

Instantiating a mutex algorithm Let us consider a n-process system. While it is possible to design mutex algorithms tailored for ad'hoc values of n (for example there are mutex algorithms specifically designed for two processes only), e.g. [50], and consequently such algorithms do not work for more than two processes. Nearly all mutex algorithms are designed to work for any value of $n \geq 2$, i.e., n is a parameter that can differ in each instance of the algorithm.[3].

On the fault-tolerance side A process crashes when it unexpectedly and definitively halts. Usual algorithms that solve mutex allow a process to crash when it is not executing acquire(), release() or the code in the critical section. Unfortunately mutual exclusion cannot solved if a process may crash at any time. This is due to the fact that if a process crashes while executing acquire(), release() or the code in the critical section, due to asynchrony, no other process can be informed of its crash. To solve this issue, the system must be enriched with additional computational power.

An approach consists in providing processes with information on failures. This is the *failure detector* approach introduced in [11]. The integer n being the number of processes, let us consider a model that allows up to t processes to crash. When considering systems where processes communicate through read/write registers, the weakest failure detector (denoted QP for Quasi-Perfect) that allows mutex to be solved has been introduced in [16]. Weakest means that no failure detector that provides processes with less information on failures than QP allows mutex to be solved in read/write systems. Assuming $t < n/2$ (i.e. the system

[3] As we will see in Section 7, due to computability issues, the situation is different for consensus where a consensus algorithm for n processes does not work for $(n + x)$ processes for $x \geq 1$. This is related to the additional computability power needed to solve consensus in crash-prone systems.

is partition-free), the weakest failure detector (denoted T) that allows mutual exclusion to solved in synchronous message-passing systems has been introduced in [15].

These two failure detectors have close but different definitions. Both are weaker that the perfect failure P and stronger than the eventually perfect failure detector $\diamond P$ defined in [11]. Moreover, both are stronger than the weakest failure detector (eventual leader denoted Ω) that allows consensus to be solved in read/write systems when $t < n$ [43] and in message-passing when $t < n/2$ [10].

Transactional memory The concept of transactional memory was proposed by M. Herlihy and J. Moss in 1993 [28], and later refined by N. Shavit and D. Touitou[56]. The idea is to provide the designers of multiprocess programs with a language construct (namely, the notion of an atomic operation called a *transaction*) that discharges them from the management of synchronization issues. More precisely, a programmer has to concentrate her efforts only on defining which parts of processes have to be executed atomically and not on the way atomicity (mutual exclusion) is realized, this last issue being automatically handled by the underlying system.

5 A Trivial Observation: Physical Objects vs Logical Objects

A *physical* object is an object that cannot be replicated by software (e.g., a printer), while a *logical* (or *immaterial*) object is an object the value of which can by replicated by software (data). Said differently, at the basic level the value of a logical object is a structured set of bits while a physical object is a hardware device.

6 1971, 1977: Once Upon a Time: the Readers/Writers Problem

A file is a logical object that provides processes with two operations: `read_file`() that allows a process to read the file and `write_file`() that allows a process to modify its content.

The Readers/Writers Problem It was observed by P. Courtois, F. Heymans, and D. Parnas [14] (1971) that mutual exclusion is stronger than necessary to provide the synchronization needed to correctly implement the `write_file`() and `read_file`() operations, namely, only each execution of `write_file`() must be

executed in mutual exclusion with any other operation execution (`read_file`() or `write_file`()), while the executions `read_file`() needs to be executed in mutual exclusion only with respect to `write_file`() (and not among themselves). As far as we know, this was the first (implicit) distinction between physical and logical objects.

The wait-freedom approach This approach was later generalized by L. Lamport to allow concurrent readings while writing [34] (1977), which showed that (as a file is a logical object) the readers/writer problem does not need mutual exclusion to be solved (see also [51]). This approach culminated in the notion of *wait-free* computing introduced by M. Herlihy [26]. Wait-free means that the progress of a process cannot be prevented by the behavior of the other processes (arbitrary unknown speed or crash failures).

From safe read/write bits to atomic read/write registers In a very interesting way, it has been shown by L. Lamport that, while the *atomicity* of basic read/write registers are sufficient to solve mutex, they are not necessary to solve it. More precisely, mutual exclusion can be solved on top of single-writer multi-reader *safe* registers (see [33, 37]). A safe register is a register that can be written by a single process and read by any number of processes. A write defines the new value of the register. A read whose execution is not concurrent with a write returns the last value written in the register. A read concurrent with a write returns *any value* that the register can contain (so it can return a value that has never been written in the register!). In a non-trivial way, multi-writer multi-reader atomic registers can be built on top of single-writer single-reader safe bits (despite asynchrony and process failures. A survey of such constructions is presented in Section V of [53].

7 1980: The Advent of Consensus: On Fault-Tolerant Distributed Computing

Definition The *consensus* problem was introduced by S. Pease, R. Shostak and L. Lamport in [41, 48] in the context of synchronous distributed systems prone to Byzantine process failures (arbitrary misbehavior of a process). This problem is at the core of distributed computing agreement problems. We consider here asynchronous read/write or message-passing systems prone to crash failures. Let a process be *correct* in a run if it does not crash during that run. In such a context a consensus object is defined by a single operation denoted propose() that takes a value as input parameter and returns a value. When a process invokes propose(v)

and obtains the value v' we say that it proposes v and decides v'. Consensus is defined by the following properties.

- Validity (safety). If a process decides v then a process proposed v.
- Agreement(safety). No two processes decide different values.
- Termination (liveness). If a process invokes propose() and does crashes, it decides.

Impossibility Unfortunately consensus is impossible to solve in the presence of asynchrony and even a single process crash, be the communication system message-passing [24] or read/write registers [44]. This means that the system has to be enriched with additional *computability power* to make consensus solvable. Several enrichments are possible.

- Enrich the system with synchrony assumptions (e.g. [23]).
- Enrich the system with scheduling assumptions (e.g. [5]).
- Enrich the system with randomization (e.g. [4, 46]).
- Restrict the set of input vectors that can be proposed (e.g. [47]). (An input vector has one entry per process containing the value it proposes. Of course a process knows only the value of its entry).
- Enrich the system with information on failures (failure detector approach [10, 11]).
- Enrich the system with asynchronous rounds such that, for each round r and each process p, the model provides the set of processes that p hears of at round r. The features of a specific system is then captured as a whole, just by a predicate over the collection of heard-of sets [12].

Consensus number of an object Let us consider an asynchronous crash prone system in which the processes communicate by reading and writing atomic registers (RW type). As just noticed, the previous impossibility results states that consensus cannot be solved in such a system. So, a fundamental question is: which additional computability power (defined not in terms of system behaviors but in terms additional object types) needs to be added to the system model so that the consensus can be solved. To this end, M. Herlihy introduced the notion of *consensus number* [26].

The consensus number of an object type T, denoted $CN(T)$, is the greatest number of processes for which consensus can be solved from any number of atomic read/write registers and any number of objects of type T. If there is no such greatest number, the consensus number of T is $+\infty$.

Let RW_TS be the type of RW registers accessed with Test&Set() operation, and RW_CS be the type of RW registers accessed with Compare&Swap() operation[4]. It has been shown in [26] that CN(RW_TS)= 2 and CN(RW_CS)= $+\infty$. More generally, [26] introduces an infinite hierarchy of objects, that cover all possible consensus numbers. The interested reader can look at [49] where is defined the notion of k-sliding window RW register. This object family spans the whole consensus hierarchy: the consensus number of the k-sliding window RW register is exactly k.

8 Consensus: a Simple Way to Agree on a Total Order

Ordering object operations Let us consider an object defined by a sequential specification, e.g., a stack with its two operations push() and pop(). To cope with asynchrony and failures, the stack (which is a logical object, i.e. a structured set of bits) is replicated on each process. So the main issue consists in ensuring that the push() and pop() operations issued by the processes are applied in the same order to all the local copies of the stack. A simple way to attain this goal consists for each process in:

1. announcing the operation it wants to execute,
2. regularly defines a sequence on the operations it sees as announced and not yet executed,
3. and proposes this sequence as input to a consensus instance.

Combined with a sequence of consensus instances (in which all processes agree a priori), this allows all the local copies of the stack to progress the same way [10, 26].

Hence, as it allows to build a total order on operations, consensus lies at the core of fault-tolerant implementations for the objects defined by a sequential specification. (For objects not defined by a sequential specification, i.e. concurrent objects, the reader can consult [7, 8, 52].)

[4]Roughly speaking both operations return the current value of the register and write a new value in it. The difference lies in the fact that Test&Set() is an unconditional write of a predefined value, while Compare&Swap() is a conditional write of a value.

Consensus vs mutex: illustration Let us consider money transfer as an object providing its users (a user is a process associated with one and only one money account) with two operations transfer() that allows a process to transfer money from its account to another account, and balance() that allows a user to read an account. Let us observe that an account is a logical object.

It has recently been shown that money transfer among a set of processes, each having its own account, does not need consensus [3, 13, 25] [5]. It is an announcement/broadcast problem that must satisfy some causality requirements.

When several persons share the same account, the associated process consists of several threads, one per person co-owner of the account. The invocations of the operations transfer() issued by the threads that are co-owners of the same account must then be ordered in order to prevent double-spending from the corresponding account. This could be realized with mutex (enriched with an appropriate failure detector or random numbers if the system is crash-prone).

But, as an account is a logical object this ordering can be realized (despite process failures and asynchrony) with the help of consensus. It follows that if each account can be accessed by at most k threads, an object the consensus number of which is k is sufficient to realize money transfer (this was first noticed in [25]).

9 Both Sides of the Same Coin

When considering objects the consistency of which is defined by a sequential specification (i.e, objects whose operations must appear as being executed sequentially), it follows from the previous simple observations that, while both mutex and consensus can be used to build a total order, mutex is for physical objects (which by nature cannot be replicated), and consensus is for logical objects (structured sets of bits which can be replicated)[6]. In this sense, mutex and consensus are the two sides of the same coin. The content of this note is summarized in Table 1.

The "Underlying coordination" column refers to the type of synchronization needed to implement mutex or consensus, namely, mutex ensures that the concerned object can be physically accessed by at most one process at a time, while consensus does not prevent several processes from invoking and simultaneously executing object operations (after these operations have been totally ordered by a consensus instance). The column "Helping needed" refers to the fact the al-

[5] It is pleasant to observe that the heavy Blockchain machinery was introduced to built a total order on the cryptocurrency operations issued by users, and this is not needed! For the interested reader, [13, 25] consider money transfer as an object defined by a sequential specification, while [3] considers money transfer as an object defined by a concurrent specification.

[6] Of course, in some specific contexts, it can be interesting to use mutex for logical objects, but this is another issue not addressed in this note.

Nature of the object	Possible replication	Total order obtained from	Underlying Coordination	Helping needed	Weakest FD
Physical	No	Mutex	strong	Yes	QP, T
Logical	Yes	Consensus	weak	Yes	Ω

Table 1: Total order: mutex vs consensus

gorithms implementing mutex or consensus need specific helping mechanisms to ensure the liveness of the operations on the object that is built [1, 9, 26, 54][7]. The last column "Weakest FD" concerns the weakest failure detectors that allows mutex or consensus to be solved. As already indicated, for read/write systems it is the failure detector QP for mutex [16] and Ω for consensus [43], while, for message-passing systems such that $t < n/2$, it is the failure detector T for mutex [15] and the eventual leader failure detector Ω for consensus [10]. It is worth noticing that the weakest information on failures that allows mutex to be solved includes a perpetual property [15, 16], while that the weakest information on failures needed to solve consensus needs to satisfy an eventual property only [10]. This is strongly related to the underlying nature of the object (physical vs logical).

Let us again insist on the fact that, in a crash-prone system where the processes communicate through read/write atomic registers (resp. message-passing when assuming $t < n/2$), the weakest failure detectors QP (resp. T) that allows mutex to be solved is stronger than the weakest failure detector Ω that allows consensus to be solved. As previously noticed, this is due to the fact that the implementation of mutex requires a stronger underlying synchronization than the one needed to implement consensus. More precisely, this is the main difference between mutex and consensus, because of their very definitions mutex does not allow concurrency at the implementation level, whereas consensus does.

Last but not least, let us notice that a recent paper by L. Lamport [40] describes a deconstruction of his famous Bakery mutex algorithm [33] from which is built a distributed state machine as defined in [35] (i.e., any object defined by a sequential specification). This can be seen as an answer to the question posed in the title of this note.

[7] As far liveness properties are concerned, wait-freedom [26] and non-blocking [29] for consensus correspond to starvation-freedom and deadlock-freedom for mutex. Differently obstruction-freedom [27] for consensus has no corresponding liveness property that could be associated with mutex (this is due to the fact that mutex implicitly considers the object to with it is applied as a "physical" object.

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] Apt K. R. and Hoare C.A.R. (editors), *Edsger Wybe Dijkstra: his life, work, and legacy*. Association for computing machinery, Morgan & Claypool Publishers, 550 pages (2022)

[3] Auvolat A., Frey D., Raynal M. and Taïani F., Money transfer made simple: a specification, a generic algorithm, and its proof. *Electronic Bulletin of EATCS (European Association of Theoretical Computer Science)*, 132:22–43 (2020)

[4] 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)

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

[6] Brinch Hansen P. (Editor), *The origin of concurrent programming*. Springer, 534 pages (2002)

[7] Castañeda A., Rajsbaum S., and Raynal M., Unifying concurrent objects and distributed tasks: Interval-linearizability. *Journal of the ACM*, 65(6), Article 45, 42 pages (2018)

[8] Castañeda A., Rajsbaum S., and Raynal M., A linearizability-based hierarchy for concurrent specifications. *Communications of the ACM*, 66(1):60–71 (2023)

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

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

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

[12] Charron-Bost and Schiper A., The heard-of model: computing in distributed systems with benign faults. *Distributed Computing*, 22:49–71 (2009)

[13] Collins D., Guerraoui R., Komatovic J., Monti M., Xygkis A., Pavlovic M., Kuznetsov P., Pignolet Y.-A., Seredinschi D.A., and Tonlikh A., Online payments by merely broadcasting messages. *Proc. 50th IEEE/IFIP Int'l Conference on Dependable Systems and Networks (DSN'20)*, pp. 26–38 (2020)

[14] Courtois P.J., Heymans F., and Parnas D.L., Concurrent control with readers and writers. *Communications of the ACM*, 14(5):667–668 (1971)

[15] Delporte-Gallet C., Fauconnier H., Guerraoui R. and Kouznetsov P., Mutual exclusion in asynchronous systems with failure detectors. *Journal of Parallel and Distributed Computing*, 65:492-505 (2005)

[16] Delporte-Gallet C., Fauconnier H., and Raynal M., On the weakest information on failures to solve mutual exclusion and consensus in asynchronous crash prone read/write systems. *Journal of Parallel and Distributed Computing*, 153:110–118 (2021)

[17] Dijkstra E.W., Over de sequentialiteit van procesbeschrijvingen (on the nature of sequential processes). *EW Dijkstra Archive (EWD-35)*, Center for American History, University of Texas at Austin (Translation by Martien van der Burgt and Heather Lawrence) (1962)

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

[19] Dijkstra E.W., Cooperating sequential processes. In *Programming Languages (F. Genuys Ed.)*, Academic Press, pp. 43-112 (1968)

[20] Dijkstra E.W., Hierarchical ordering of sequential processes. *Acta Informatica*, 1(1):115–138 (1971)

[21] Dijkstra E.W., Guarded commands, non-determinacy and formal derivation of programs. *Communications of the ACM*, 8:453–457 (1975)

[22] Dijkstra E.W., Dahl O.-J., and Hoare C.A.R., Structured programming. *Academic Press*, 220 pages (1972)

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

[24] 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)

[25] Guerraoui R., Kuznetsov P., Monti M.,Pavlovic M., Seredinschi D.A., The consensus number of a cryptocurrency. *Distributed Computing*, 35:1–15 (2022)

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

[27] 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)

[28] Herlihy M.P. and Moss J.E.B., Transactional memory: architectural support for lock-free data structures. *Proc. 20th ACM Int'l Symposium on Computer Architecture (ISCA'93)*, ACM Press, pp. 289–300 (1993)

[29] 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)

[30] Hoare C.A.R., An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580 (1969)

[31] Hoare C.A.R., Programming: sorcery or science? *IEEE Software*, 1(2):5–16 (1984)

[32] Jones C.B. and Misra J.(editors), *Theories of programming: the life and works of Tony Hoare*, Association for computing machinery, Morgan & Claypool Publishers, 430 pages (2021)

[33] Lamport L., A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455 (1974)

[34] Lamport L., Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811 (1977)

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

[36] Lamport L., On inter-process communications, part I: basic formalism. *Distributed Computing*, 1(2):77–85 (1986)

[37] Lamport L., On inter-process communications, part II: algorithms. *Distributed Computing*, 1(2):86–101 (1986)

[38] Lamport L., The mutual exclusion problem: Part I- a theory of interprocess communication. *Journal of the ACM*, 33(2): 313–326 (1986)

[39] Lamport L., The computer science of concurrency: the early years (Turing lecture). *Communications of the ACM*, 58(6):71–76 (2015)

[40] Lamport L., Deconstructing the Bakery to build a distributed state machine. *Communications of the ACM*, 65(9):58–66 (2022)

[41] Lamport L., Shostak R., and Pease S., The Byzantine generals problem, *ACM Transactions on Programming Languages and Systems*, 4(3):382–401 (1982)

[42] Lewis H.R. and Papadimitriou C.H., *Elements of the theory of computation*, Prentice Hall Int. Editions, 361 pages (1998)

[43] Lo W.K. and Hadzilacos V., Using failure detectors to solve consensus in asynchronous shared memory systems. *Proc. 8th Int'l Workshop on Distributed Algorithms (WDAG'94)*, Springer LNCS 857, pages 280–295 (1994)

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

[45] Malkhi D. (Editor), *Concurrency: the works of Leslie Lamport.* Association for computing machinery, Morgan & Claypool Publishers, 345 pages (2019)

[46] 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 the ACM*, 62(4), Article 31, 21 pages, (2015)

[47] 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)

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

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

[50] Peterson G.L., Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.

[51] Peterson G.L., Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5:46-55 (1983)

[52] Rajsbaum S. and Raynal M., Mastering concurrent computing through sequential thinking. *Communications of the ACM*, Vol. 63(1):78–87 (2020)

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

[54] Raynal M., *Fault-tolerant message-passing distributed systems: an algorithmic approach.* Springer, 550 pages (2018)

[55] Raynal M., and Taubenfeld G., A visit to mutual exclusion in seven dates. *Theoretical Computer Science*, 919:47–65 (2022)

[56] Shavit N. and Touitou D., Software transactional memory. *Distributed Computing*, 10(2):99-116 (1997)

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

[58] Taubenfeld G., Concurrent programming, mutual exclusion. *Springer Encyclopedia of Algorithms*, pp. 421–425 (2016)