Extending the Causal Consistency Condition to any Object Defined by a Sequential Specification

Matthieu Perrin, Achour Mostefaoui, Michel Raynal

Extending the Causal Consistency Condition to any Object Defined by a Sequential Specification

Achour Mostéfaoui[†], Matthieu Perrin[†], Michel Raynal[‡,⋆]

[†]LINA, Université de Nantes, 44322 Nantes, France
[‡]Univ Rennes, IRISA, 35042 Rennes, France
[⋆]Department of Computing, Polytechnic University, Hong Kong

Abstract

This paper presents a generalization of causal consistency suited to the family of objects defined by a sequential specification. As causality is captured by a partial order on the set of operations issued by the processes on shared objects (concurrent operations are not ordered), it follows that causal consistency allows different processes to have different views of each object history.

Keywords: Causality, Causal order, Concurrent object, Consistency condition.

1 Processes and Concurrent Objects

Let us consider a set of n sequential asynchronous processes p_1, ..., p_n, which cooperate by accessing shared objects. These objects are called *concurrent* objects. A main issue consists in defining the correct behavior of concurrent objects. Two classes of objects can be distinguished according to way they are specified.

- The objects which can be defined by a sequential specification. Roughly speaking, this class of objects includes all the objects encountered in sequential computing (e.g., queue, stack, set, dictionary, graph). Different tools can be used to define their correct behavior (e.g., transition function, list of all the correct traces -histories-, pre and post-conditions, etc.).

It is usually assumed that the operations accessing these objects are *total*, which means that, whatever the current state of the object, an operation always returns a result.

As an example, let us consider a bounded stack. A pop() operation returns a value if the stack is not empty, and returns the value $\perp$ if it is empty. A push(v) operation returns the value $\top$ if the stack is full, and returns ok otherwise (v was then added to the stack). A simpler example is a read/write register, where a read operation always returns a value, and a write operation always returns ok.

- The objects which cannot be defined by a sequential specification. Example of such objects are Rendezvous objects or Non-blocking atomic commit objects [10]. These objects require processes to wait each other, and their correct behavior cannot be captured by sequences of operations applied to them.

In the following we consider objects defined by a sequential specification.

2 Strong Consistency Conditions

Strong consistency conditions are *natural* (and consequently easy to understand and use) in the sense that they require each object to appear as if it has been accessed sequentially. In a failure-free context, this can be easily obtained by using mutual exclusion locks bracketing the invocation of each operation.

Atomicity/Linearizability The most known and used consistency condition is *atomicity*, also called *linearizability*[1]. It requires that each object appears as if it was accessed sequentially, this sequence of operations belonging to the specification of the object, and complying with the real-time order of their occurrences.

Sequential consistency This consistency condition, introduced in [15], is similar to, but weaker than, linearizability, namely, it does not require the sequence of operations to comply with real-time order.

[1]Atomicity was formally defined in [16, 17] for basic read/write objects. It was then generalized to any object defined by a sequential specification in [13]. We consider these terms as synonyms in the following.

Figure 1 presents an example of a sequentially consistent computation (which is not atomic) involving two read/write registers $R1$ and $R2$, accessed by two processes p_1 and p_2. The dashed arrows define the *causality* relation linking the read and write operations on each object (also called *read-from* relation when the object is a read/write register). It is easy to see that the sequence of operations made up of all the operations issued by p_2, followed by all the operations issued by p_1, satisfies the definition of sequential consistency.

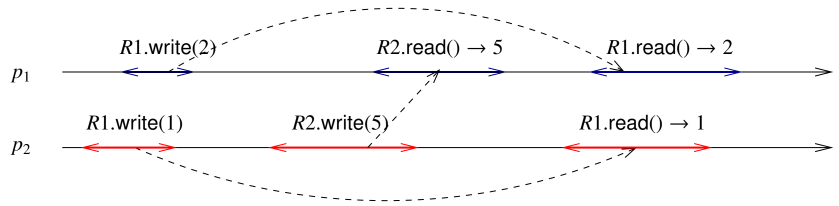

Figure 1: A sequentially consistent computation (which is not atomic)

Implementing a strong consistency condition in an asynchronous message-passing system Shared memories usually provide processes with objects built on top of basic atomic read/write objects or more sophisticated objects accessed by atomic operations such as Test&Set or Compare&Swap [11, 12, 22, 25]. This is no longer the case in message-passing systems where all the objects (except communication channels) have to be built from scratch [5, 21].

Implementations of sequentially consistent objects and atomic objects in failure-free message-passing systems can be found in [4, 5, 7, 20, 21]. These implementations rest on a mechanism which allows a total order on all operations to be built. This can be done by a central server, or a broadcast operation delivering messages in the same order at all the processes. Such an operation is usually called *total order broadcast* (TO-broadcast) or *atomic broadcast*. It is shown in [20] that, from an implementation point of view, sequential consistency can be seen as a form of lazy linearizability. The "compositional" power of sequential consistency is addressed in [8, 18].

Implementations of a strong consistency condition (such as atomicity) in failure-prone message-passing systems is more difficult. More precisely, except for a few objects including read/write registers (which can be built only in systems where, in each execution, a majority of processes do not crash [3]), it is impossible to implement an atomic object in the presence of asynchrony and process crashes [9]. Systems have to be enriched with additional computing power (such as randomization or failure detectors) to be able to implement objects defined by a strong consistency condition.

3 Causal Consistency on Read/Write Objects (Causal Memory)

Causality-based consistency condition A causal memory is a set of read/write objects satisfying a consistency property weaker that atomicity or sequential consistency. This notion was introduced in [1]. It relies on a notion of *causality* similar to the one introduced in [14] for message-passing systems.

The main difference between causal memory and the previous strong consistency conditions lies in the fact that causality is captured by a partial order, which is trivially weaker than a total order. A total order-based consistency condition forces all the processes to see the same order on the object operations. Causality-based consistency does not. Each process can have its own view of the execution, their "greatest common view" being the causality partial order produced by the execution. Said differently, an object defined by a strong consistency condition is a *single-view* object, while an object defined by a causality-based consistency condition is a *multi-view* object (one view per process).

Another difference between a causality-based consistency condition and a strong consistency condition lies in the fact that a causality-based consistency condition copes naturally with process crashes and system partitioning.

Preliminary definitions As previously indicated, a causal memory is a set of read/write registers. Its semantics is based on the following preliminary definitions (from [1, 13]). To simplify the presentation and without loss of generality, we assume that (a) all the values written in a register are different, and (b) each register has an initial value written by a fictitious write operation.

- A *local (execution) history* L_i of a process p_i is the sequence of read and write operations issued by this process. If the operations $\mathsf{op}1$ and $\mathsf{op}2$ belong to L_i and $\mathsf{op}1$ appears before $\mathsf{op}2$, we say "$\mathsf{op}1$ precedes $\mathsf{op}2$ in p_i's process order". This is denoted $\mathsf{op}1 \xrightarrow{i} \mathsf{op}2$.
- The *write-into relation* (denoted $\xrightarrow{wi}$) captures the effect of write operations on the read operations. Denoted $\xrightarrow{wi}$, it is defined as follows: $\mathsf{op}1 \xrightarrow{wi} \mathsf{op}2$ if $\mathsf{op}1$ is the write of a value v into a register R and $\mathsf{op}2$ is a read operation of the register R which returns the value v.
- An *execution history* H is a partial order composed of one local history per process, and a partial order, denoted $\xrightarrow{po}$, defined as follows: $\mathsf{op}1 \xrightarrow{po} \mathsf{op}2$ if
  - $\mathsf{op}1, \mathsf{op}2 \in L_i$ and $\mathsf{op}1 \xrightarrow{i} \mathsf{op}2$ (process order), or
  - $op1 \xrightarrow{wi} op2$ (write-into order), or

- $\exists$ op3 such that op1 $\xrightarrow{po}$ op3 and op3 $\xrightarrow{po}$ op2 (transitivity).

- Two operations not related by $\xrightarrow{po}$ are said to be *independent* or *concurrent*.
- The projection of H on a register R (denoted $H|R$) is the partial order H from which are suppressed all the operations which are not on R.
- A *serialization* S of an execution history H (whose partial order is $\xrightarrow{po}$) is a total order such that, if op1 $\xrightarrow{po}$ op2, then op1 precedes op2 in S.

A remark on the partial order relation As we can see, the read-from relation mimics the causal send/receive relation associated with message-passing [14]. The difference is that zero, one, or several reads can be associated with the same write. In both cases, the (write-into or message-passing) causality relation is a global property (shared by all processes) on which is built the consistency condition. It captures the effect of the environment on the computation (inter-process asynchrony), while process orders capture the execution of the algorithms locally executed by each process.

Causal memory Let H_{i+w} be the partial order $\xrightarrow{po}$, from which all the read operations not issued by p_i are suppressed (the subscript $i + w$ means that only all the operations issued by p_i plus all write operations are considered).

As defined in [1], an execution history H is *causal* if, for each process p_i, there is a serialization S_i of H_{i+w} in which each read from a register R returns the value written in R by the most recent preceding write in R.

This means that, from the point of view of each process p_i, taken independently from the other processes, each register behaves as defined by its sequential specification. It is important to see, that different processes can have different views of a same register, each corresponding to a particular serialization of the partial order $\xrightarrow{po}$ from which the read operations by the other processes have been eliminated.

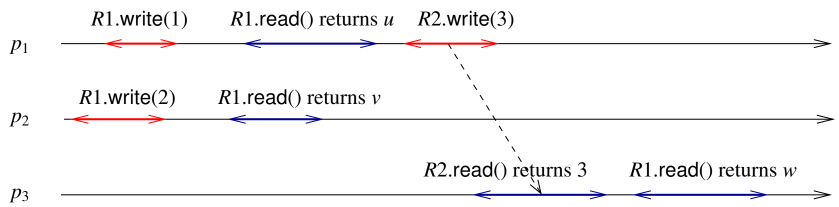

Figure 2: Example of an execution of a causal read/write memory

An example of a causal memory execution is depicted in Figure 2. Only one write-into pair is indicated (dashed arrow). As $R1$.write(1) and $R1$.write(2) are independent, each of the operations $R1.\mathsf{read}()$ by p_2 and p_3 can return any value, i.e., $u, v \in \{1, 2\}$. For the same reason, and despite the write-into pair on the register $R2$ involving p_1 and p_3, the operation $R1.\mathsf{read}()$ issued by p_3 can return $w \in \{1, 2\}$. This shows that different processes can obtain different "views" of the same causal memory execution. Once a read returned a value, a new write-into pair is established.

Implementations of a causal read/write memory (e.g., [2]) rest on an underlying communication algorithm providing causal message delivery [6, 24]. It is shown in [1, 23] that, in executions that are data race-free or concurrent write-free, a causal memory behaves as a sequentially consistent read/write memory.

4 Causal Consistency for any Object

The problem Albeit it was introduced more than 20 years ago, it appears that, when looking at the literature, causal consistency has been defined and investigated only for read/write objects (the only exception we are aware of is [19]). This seems to be due to the strong resemblance between read/write operations and send/receive operations. Hence, the question: Is it possible to generalize causal consistency to any object defined by a sequential specification? This section answers positively this question.

Preliminary definitions The notations and terminology are the same as in the previous section, but now the operations are operations on any object O of a set of objects $\mathcal{O}$, each defined by a sequential specification.

Considering a set of local histories and a partial order $\xrightarrow{po}$ on their operations, let $Assignment_i(\xrightarrow{po})$ denote the partial order $\xrightarrow{po}$, in which, for each operation $\mathsf{op}()$ not issued by p_i, the returned value v is replaced by a value v', possibly different from v, the only constraint being that v and v' belong to the same domain (as defined by the corresponding operation $\mathsf{op}()$). Let us notice that $Assignment_i(\xrightarrow{po}$) is not allowed to modify the values returned by the operations issued by p_i. Moreover, according to the domain of values returned by the operations, a lot of different assignments can be associated with each process p_i.

Given a partial order $\xrightarrow{po}$, and an operation op, the *causal past* of op with respect to $\xrightarrow{po}$ is the set of operations $\{\mathsf{op}' \mid \mathsf{op}' \xrightarrow{po} \mathsf{op}\}$. A serialization S_i of a partial order $\xrightarrow{po}$ is said to be *causal past-constrained* if it is such that, for any operation op issued by p_i, only the operations of the causal past of op appear before op.

Causal consistency for any object Let $H = \langle L_1, \ldots, L_n \rangle$ be a set of n local histories (one per process) which access a set $\mathcal{O}$ of concurrent objects, each defined by a sequential specification. H is *causally consistent* if there is a partial order $\xrightarrow{po}$ on the operations of H such that for any process p_i:

- $(\mathsf{op1} \xrightarrow{i} \mathsf{op2}) \Rightarrow (\mathsf{op1} \xrightarrow{po} \mathsf{op2})$, and
- $\exists$ an assignment $Assignment_i$ and a causal past-constrained serialization S_i of $Assignment_i(\xrightarrow{po})$ such that, $\forall\ O \in \mathcal{O}$, $S_i|O$ belongs to the sequential specification of O.

The first requirement states that the partial order $\xrightarrow{po}$ must respect all process orders. The second requirement states that, as far as each process p_i is concerned, the local view (of $\xrightarrow{po}$) it obtains is a total order (serialization S_i) that, according to some value assignment, satisfies the sequential specification of each object O.[2]

Let us remark that the assignments $Assignment_i()$ and $Assignment_j()$ associated with p_i and p_j, respectively, may provide different returned values in S_i and S_j for the same operation. Each of them represents the local view of the corresponding process, which is causally consistent with respect to the global computation as captured by the relation $\xrightarrow{po}$.

When the objects are read/write registers The definition of a causal memory stated in Section 3 is a particular instance of the previous definition. More precisely, given a process p_i, the assignment $Assignment_i$ allows an appropriate value to be associated with every read not issued by p_i. Hence, there is a (local to p_i) assignment of values such that, in S_i, any read operation returns the last written value. In a different, but equivalent way, the definition of a causal read/write memory given in [1] eliminates from S_i the read operations not issued by p_i.

While such operation eliminations are possible for read/write objects, they are no longer possible when one wants to extend causal consistency to any object defined by a sequential specification. This come from the observation that, while a write operation resets "entirely" the value of the object, "update" operations on more sophisticated objects defined by a sequential specification (such as the operations $\mathsf{push}()$ and $\mathsf{pop}()$ on a stack for example), do not reset "entirely" the value of the object. The memory of such objects has a richer structure than the one of a basic read/write object.

[2]This definition is slightly stronger than the definition proposed in [19]. Namely, in addition to the introduction of the assignment notion, the definition introduced above adds the constraint that, if an operation op precedes an operation op' in the process order, then the serialization required for op must be a prefix of the serialization required for op'. On the other hand, it describes precisely the level of consistency achieved by Algorithm 1 presented below.

An example As an example illustrating the previous general definition of a causally consistent object, let us consider three processes p_1 p_2 and p_3, whose accesses to a shared unbounded stack are captured by the following local histories L_1, L_2, and L_3. In these histories, the notation $\mathsf{op}_i(a)r$ denotes the operation $\mathsf{op}()$ issued by p_i, with the input parameter a, and whose returned value is r.

- $L_1 = \mathsf{push}_1(a)\mathsf{ok},\ \mathsf{push}_1(\mathsf{c})\mathsf{ok}, \mathsf{pop}_1()c.$
- $L_2 = \mathsf{pop}_2()a,\ \mathsf{push}_2(b)\mathsf{ok}, \mathsf{pop}_2()b.$
- $L_3 = \mathsf{pop}_3()a,\ \mathsf{pop}_3()b.$

Hence, the question: Is $H = \langle L_1, L_2, L_3 \rangle$ causally consistent? We show that the answer is "yes". To this end we need first to build a partial order $\xrightarrow{po}$ respecting the three local process orders. Such a partial order is depicted in Figure 3, where process orders are implicit, and the inter-process causal relation is indicated with dashed arrows (let us remind that this relation captures the effect of the environment –asynchrony– on the computation).

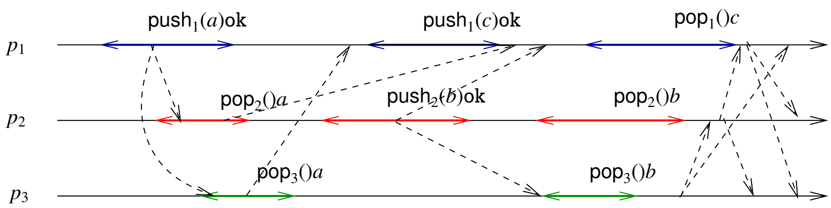

Figure 3: Example of a partial order on the operations issued on a stack

The second step consists in building three serializations respecting $\xrightarrow{po}$, S_1 for p_1, S_2 for p_2, and S_3 for p_3, such that, for each process p_i, there is an assignment of values returned by the operations $\mathsf{pop}()$ ($Assignment_i()$), from which it is possible to obtain a serialization S_i belonging to the specification of the stack. Such assignments/serializations are given below.

- $S_1 = \mathsf{push}_1(a)\mathsf{ok},\ \mathsf{pop}_3()a,\ \mathsf{push}_1(c)\mathsf{ok},\ \mathsf{pop}_2()\bot,\ \mathsf{push}_2(b)\mathsf{ok},$
  $\mathsf{pop}_1()c,\ \mathsf{pop}_2()b,\ \mathsf{pop}_3()\bot.$

- $S_2 = \mathsf{push}_1(a)\mathsf{ok},\ \mathsf{pop}_2()a,\ \mathsf{push}_2(b)\mathsf{ok},\ \mathsf{pop}_2()b,\ \mathsf{pop}_3()\bot,$
  $\mathsf{pop}_3()\bot, \mathsf{push}_1(c)\mathsf{ok},\ \mathsf{pop}_1()c.$

- $S_3 = \mathsf{push}_1(a)\mathsf{ok},\ \mathsf{pop}_3()a,\ \mathsf{pop}_2()\bot,\ \mathsf{push}_2(b)\mathsf{ok},\ \mathsf{pop}_3()b,$
  $\mathsf{pop}_2()\bot,\ \mathsf{push}_1(c)\mathsf{ok},\ \mathsf{pop}_1()c.$

The local view of the stack of each process p_i is constrained only by the causal order depicted in Figure 3, and also depends on the way it orders concurrent operations. As far as p_2 is concerned we have the following, captured by its serialization/assignment S_2. (The serializations S_1 and S_3 are built similarly.) We have considered short local histories, which could be prolonged by adding other operations. As depicted in the figure, due to the last causality (dashed) arrows, those operations would have all the operations in $L_1 \cup L_2 \cup L_3$ in their causal past.

1. Process p_2 sees first $\mathsf{push}_1(a)\mathsf{ok}$, and consequently (at the implementation level) updates accordingly its local representation of the stack.
2. Then, p_2 sees its own invocation of $\mathsf{pop}_2()$ which returns it the value a.
3. Then, p_2 sees its own $\mathsf{push}_2(b)$ and $\mathsf{pop}_2()$ operations; $\mathsf{pop}_2()$ returns consequently b.
4. Finally p_2 becomes aware of the two operations $\mathsf{pop}_3()$ issued by p_3, and the operations $\mathsf{push}_1(c)$ and $\mathsf{pop}_1()$ issued by p_1. To have a consistent view of the stack, it considers the assignment of returned values that assigns the value $\perp$ to the two operations $\mathsf{pop}_3()$, and the value c to the operations $\mathsf{pop}_1()$. In this way, p_2 has a consistent view of the stack, i.e., a view which complies with the sequential specification of a stack.

A universal construction Algorithm 1 is a universal construction which builds causally consistent objects from their sequential specification. It considers deterministic objects. This algorithm is built on top of any underlying algorithm ensuring causal broadcast message delivery [6, 24][3]. Let "co_broadcast MSG(a)" denote the causal broadcast of a message tagged MSG carrying the value a. The associated causal reception at any process is denoted "co-delivery". "?" denotes a control value unknown by the processes at the application level.

when p_i **invokes** $O.\mathsf{op}(param)$ **do**
(1) $result_i \leftarrow ?$;
(2) co_broadcast OPERATION$(i, O, \mathsf{op}(param))$;
(3) wait $(result_i \neq ?)$;
(4) return $(result_i)$.

when OPERATION$(j, O, \mathsf{op}(param))$ **is** co-delivered **do**
(5) $\langle r, state_i[O]\rangle \leftarrow \delta_O(state_i[O], \mathsf{op}(param))$;
(6) **if** $(j = i)$ **then** $result_i \leftarrow r$ **end if**.

Algorithm 1: Universal construction for causally consistent objects (code for p_i)

[3]Interestingly, the replacement of the underlying message causal order broadcast by a message total order broadcast, implements linearizability.

Each object O is defined by a transition function $\delta_O()$, which takes as input parameter the current state of O and the operation $\mathsf{op}(param)$ applied to O. It returns a pair $\langle r, new\_state \rangle$, where r is the value returned by $\mathsf{op}(param)$, and new_state is the new state of O. Each process p_i maintains a local representation of each object O, denoted $state_i[O]$.

When a process p_i invokes an operation $\mathsf{op}(param)$ on an object O, it co-broadcasts the message OPERATION$(i, O, \mathsf{op}(param))$, which is co-delivered to each process (i.e., according to causal message order). Then, p_i waits until this message is locally processed. When this occurs, it returns the result of the operation.

When a process p_i co-delivers a message OPERATION$(j, O, \mathsf{op}(param))$, it updates accordingly its local representation of the object O. If p_i is the invoking process, it additionally locally returns the result of the operation.

5 Conclusion

This short article extended the notion of causal consistency to any object defined by a sequential specification. This definition boils down to causal memory when the objects are read/write registers.

The important point in causal consistency lies in the fact that each process has its own view of the objects, and all these views agree on the partial order on the operations but not necessarily on their results. More explicitly, while each process has a view of each object, which locally satisfies its object specification, two processes may disagree on the value returned by some operations. This seems to be the "process-to-process inconsistency cost" that must be paid when weakening consistency by considering a partial order instead of a total order. On another side and differently from strong consistency conditions, causal consistency copes naturally with partitioning and process crashes.

Acknowledgments

This work has been partially supported by the Franco-German DFG-ANR Project 40300781 DISCMAT (devoted to connections between mathematics and distributed computing), and the French ANR project DESCARTES (devoted to layered and modular structures in distributed computing).

References

[1] Ahamad M., Neiger G., Burns J.E., Hutto P.W., and Kohli P., Causal memory: definitions, implementation and programming. *Distributed Computing*, 9:37-49 (1995)

[2] Ahamad M., Raynal M. and Thia-Kime G., An adaptive protocol for implementing causally consistent distributed services. *Proc. 18th Int'l Conference on Distributed Computing Systems (ICDCS'98)*, IEEE Press, pp. 86-93 (1998)

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

[4] Attiya H. and Welch J.L., Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 12(2):91-122 (1994)

[5] Attiya H. and Welch J., *Distributed computing: fundamentals, simulations and advanced topics*, (2d Edition), Wiley-Interscience, 414 pages (2004)

[6] Birman K.P. and Joseph T.A., Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47-76 (1987)

[7] Cholvi V., Fernández A., Jiménez E., Manzano P., and Raynal M., A methodological construction of an efficient sequentially consistent distributed shared memory. *The Computer Journal*, 53(9):1523-1534 (2010)

[8] Ekström N. and Haridi S., A fault-tolerant sequentially consistent DSM with a compositional correctness proof. *Proc. 4th Int'l Conference on Networked Systems (NETYS'16)*, Springer LNCS 9944, pp. 183-192 (2016)

[9] 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)

[10] Gray J., Notes on database operating systems: an advanced course. *Springer LNCS* 60, pp. 10-17 (1978)

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

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

[13] 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)

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

[15] Lamport L., How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C28(9):690–691 (1979)

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

[17] Misra J., Axioms for memory access in asynchronous hardware systems. *ACM Transactions on Programming Languages and Systems*, 8(1):142-153 (1986)

[18] Perrin M., Petrolia M., Mostéfaoui A., and Jard Cl., On composition and implementation of sequential consistency. *Proc. 30th Int'l Symposium on Distributed Computing (DISC'16*, Springer LNCS 9888, pp. 284-297 (2016)

[19] Perrin M., Mostéfaoui A., and Jard Cl., Causal consistency: beyond memory. *Proc. 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'16)*, ACM Press, Article 26, 12 pages (2016)

[20] Raynal M., Sequential consistency as lazy linearizability. *Proc. 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA'02)*, ACM press, pp. 151-152 (2002)

[21] Raynal M., *Distributed algorithms for message-passing systems*. Springer, 510 pages, ISBN 978-3-642-38222-5 (2013)

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

[23] Raynal M. and Schiper A., From causal consistency to sequential consistency in shared memory systems. *Proc. 15th Int'l Conference on Foundations of Software Technology and Theoretical Computer Science (FST&TCS)*, Springer LNCS 1026, pp. 180-194 (1995)

[24] Raynal M., Schiper A. and Toueg S., The causal ordering abstraction and a simple way to implement it. *Information Processing Letters*, 39(6):343-350 (1991)

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