A Simple Object that Spans the Whole Consensus Hierarchy

Achour Mostefaoui, Matthieu Perrin, Michel Raynal

A Simple Object that Spans the Whole Consensus Hierarchy

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 simple generalization of the basic atomic read/write register object, whose genericity parameter spans the whole set of integers and is such that its k-parameterized instance has exactly consensus number k. This object, whose definition is natural, is a sliding window register of size k. Its interest lies in its simplicity and its genericity dimension which provides a global view capturing the whole consensus hierarchy. Hence, this short article should be seen as a simple pedagogical introduction to Herlihy's consensus hierarchy.

Keywords: Asynchronous system, Atomic read/write register, Consensus number, Consensus, Distributed computability, Generic object type, Herlihy's (consensus) hierarchy, Ledger object, Process crash failure.

1 Wait-free Computing Model and the Consensus Hierarchy

Crash-prone asynchronous read/write-based systems This paper considers the classical distributed computing model called *read/write wait-free* model [6]. It is composed of a set of n sequential processes denoted p_1, ..., p_n, which communicate through atomic read/write registers [7, 8, 11, 14].

Each process is asynchronous, which means that it proceeds at its own speed, which can be arbitrary and remains always unknown to the other processes, and executes its local algorithm until it possibly crashes, where a crash is a premature halt. Any number of processes may crash in a run, and after crashing a process does not recover. A process that crashes in a run is said to be *faulty*. Otherwise, it is *correct* or *non-faulty*. Let us notice that, due to process crashes and asynchrony, no process can know if another process crashed or is only very slow.

Consensus object The notion of a *universal* object with respect to fault-tolerance was introduced by M. Herlihy [6]. An object type T is *universal* if it is possible to implement any object (defined by a sequential specification) in the read/write wait-free model enriched with any number of objects of type T. An algorithm providing such an implementation is called a *universal construction*. It is shown in [6] that *consensus* objects are universal. These objects allow the processes to propose values and agree on one of them. More precisely, such an object provides the processes with a single operation, denoted $\mathsf{propose}()$, that a process can invoke only once, and returns it a value. When p_i invokes $\mathsf{propose}(v_i)$,we say that it "proposes the value v_i", and if v is the returned value we say that it "decides v". The consensus object is defined by the three following properties:

- Validity. If a process decides a value, this value was proposed by a process.
- Agreement. No two processes decide different values.

- Termination. If a correct process invokes propose(), it decides a value.

Termination states that if a correct process invokes propose(), it decides a value whatever the behavior of the other processes (wait-freedom progress condition). Validity connects the output to the inputs, while Agreement states that the processes cannot decide differently. A sequence of consensus objects is used in the following way in a universal construction. According to its current view of the operations invoked on, and not yet applied to, the object O of type T that is built, each process proposes to the next consensus instance a sequence of operations to be applied to O, and the winning sequence is actually applied. An helping mechanism [3] is used to ensure that all the operations on O by any correct process are eventually applied to O.

Consensus numbers and consensus hierarchy The notion of a *consensus number* associated with an object type T (denoted CN(T) in the following) was introduced by Herlihy in [6]. It is the greatest positive integer n such that consensus can be implemented in a system of n processes with atomic read/write registers and objects of type T. If there is no such finite n, the consensus number of T is $+\infty$. Hence, a type T such that CN$(T) \geq n$ is universal in a system of n (or less) processes.

It appears that the consensus numbers define an infinite hierarchy (Herlihy's hierarchy) in which atomic read/write registers have consensus number 1, object types such as Test&Set, Fetch&Add, and Swap, have consensus number 2, etc., until object types such as Compare&Swap, Linked Load/Store Conditional (and a few others) that have consensus number $+\infty$. In between, read/write registers provided with m-assignment[1] with $m > 1$, have consensus number $(2m-2)$. (Recent developments on synchronization objects and consensus numbers can be found in [1, 3, 9].)

Content of the paper This paper addresses the following question: Does it exist a simple object family, parameterized by a positive integer k, that covers the whole consensus hierarchy (i.e., whose object instantiated with number k has exactly consensus number k)? The paper answers positively this question by presenting a simple object family, and shows that, for any $k \geq 1$, its k-parameterized instance has consensus number k. This object is a very simple and natural generalization of the most basic shared object, namely the atomic read/write register, extended to become a sliding window register of size k. This object family has two noteworthy properties. One is its simplicity. The other one lies in the fact that (to our knowledge) it is the only generic object spanning *all* consensus numbers. This has several advantages, among which, its pedagogical dimension (easy to understand and teach to students), its universality dimension (no need to introduce a specific object at each level of the consensus hierarchy to capture it), and its definition itself (a simple and natural generalization of an atomic read/write register). As an immediate consequence of this result, a short Appendix shows that the consensus number of the ledger object (such as the one used in cryptocurrencies) is $+\infty$.

2 The Atomic k-Sliding Read/Write Register (RW$_k$)

Definition As previously indicated, a *k-sliding read/write register* (in short RW$_k$) is a natural generalization of an atomic read/write register, which corresponds to the case $k = 1$. Let $KREG$ be such an object. It can be seen as a sequence of values, accessed by two atomic operations denoted $KREG$.write() and $KREG$.read(). "Atomic" means that these operations appear as if they have been executed in some sequential order, and this total order is such that, if operation op1 terminates before operation op2 starts, then op1 appears before op2 [8, 11, 14].

[1]Such an assignment updates atomically m read/write registers. It is sometimes written $X_1, X_2, \cdots, X_m \leftarrow v_1, \cdots, v_m$ where the X_i are the registers, and each v_i the value assigned to X_i.

The invocation of $KREG.\mathsf{write}(v)$ by a process adds the value v at the end of the sequence $KREG$, while an invocation of $KREG.\mathsf{read}()$ returns the ordered sequence of the last k written values (if only $x < k$ values have been written, the default value $\perp$ replaces each of the $(k - x)$ missing values).

Hence, an RW$_k$ object is a sequence containing all the values that have been written (in their atomicity-defined writing order), and whose each read operation returns the k values that have been written just before it, according to the atomicity order. As already indicated, it is easy to see that, for $k = 1$, RW$_k$ is a classical atomic read/write register. For $k = +\infty$, each read operation returns the whole sequence of values written so far. Let us notice that RW$_k$ objects appear in some applications (e.g., the object that models the content of a screen in an email service where only the last k received messages are displayed, or the screen describing plane time departures in airports [16]).[2]

Ranking the objects of the $\{$RW$_k\}_{k\geq 1}$ family Let RW$_k \geq$ RW$_{k'}$ denotes the fact that an RW$_{k'}$ object can be built from an RW$_k$ object. The following property follows directly the length of the sequences returned by these objects.

Property 1 $\forall k, k' : (k \geq k') \Rightarrow$ (RW$_k \geq$ RW$_{k'}$).

3 The Consensus Number of RW$_k \geq k$

This section shows that the consensus number of an RW$_k$ object is at least k. To this end, Algorithm 1 builds a consensus object for k processes from an RW$_k$ object $KREG$.

operation $\mathsf{propose}(v_i)$ **is**
(1) $KREG.\mathsf{write}(v_i)$
(2) $seq_i \leftarrow KREG.\mathsf{read}()$;
(3) **let** d **be** the first non-$\perp$ value in seq_i;
(4) $\mathsf{return}(d)$
end operation.

Algorithm 1: Solving consensus from an RW$_k$ object (code for p_i)

Theorem 1 *For any positive integer k we have* CN(RW$_k$) $\geq k$.

Proof Let us consider a read/write wait-free system of k processes. The consensus Termination property follows from the Termination properties of the operations $KREG.\mathsf{write}()$ and $KREG.\mathsf{read}()$ of the underlying atomic object $KREG$ (lines 1 and 2), and the fact that the algorithm contains neither loops, nor wait statements.

As at most k processes invoke the consensus operation $\mathsf{propose}()$, the underlying object $KREG$ contains at most k values. Moreover, the oldest of them is the value v written by the first process that executed $KREG.\mathsf{write}()$ (line 1). It follows that the value extracted (line 3) from its local sequence seq_i by any process p_i is v, which proves the consensus Agreement property. The proof of the consensus Validity property follows from the same reasoning. $\Box_{Theorem\ 1}$

4 The Consensus Number of RW$_k \leq k$

This section shows that, for any finite value k, the consensus number of an RW$_k$ object is smaller than $(k + 1)$. The proof is a simple adaptation of impossibility proofs found in textbooks (such as [2, 13, 17,

[2]An object close to RW$_k$ objects was concurrently and independently introduced in [4] to address complexity issues in the context of multiprocessor synchronization.

19]), which all rest on the basic concepts (e.g., notion of valence) and techniques introduced in [5] in the context of message-passing systems and then used in [12] in the context of wait-free read/write systems.

Definitions (The definitions that follow are from [5].) Without loss of generality, the proof considers binary consensus, i.e., only the values 0 and 1 can be proposed by the processes (there are algorithms that implement multivalued consensus on top of binary consensus [17]).

A configuration is a global state made up of the local states of each process and the state of every object shared by the processes. In our case, as RW$_k \geq$ RW$_1$(Property 1), we consider that the only objects shared by the processes are RW$_k$ objects.

Assuming an algorithm A implementing a consensus object, a configuration Σ attained by an execution of A is v-valent ($v \in \{0,1\}$), if only the value v can be decided from Σ. Such configurations are said to be *monovalent*. Otherwise, they are said to be *bivalent* (the dices are not yet cast!). Let us observe that there is an initial configuration that is bivalent[3]. Moreover, let us notice that -due to its very definition- any configuration that follows a v-valent configuration is v-valent.

A schedule σ is a sequence of operations on shared objects issued by the processes. Let us observe that, given an initial configuration, any consensus algorithm A must terminate (all correct processes must decide). Consequently all the schedules it can produce (whatever the failure and asynchrony pattern) must eventually attain a monovalent configuration.

Σ being a configuration, let $\mathsf{op}_x(\Sigma)$ denotes the configuration attained from Σ by executing op_x (the next read or write operation on a RW$_k$ object issued by p_x), and $\sigma(\Sigma)$ be the configuration attained from Σ by executing the schedule σ.

A *maximal* bivalent schedule is a schedule that ends in a bivalent configuration Σ such that the next operation issued by any process produces a monovalent configuration. Let us notice that, if there is an algorithm solving consensus, any of its executions has a maximal schedule (otherwise A will have non-terminating executions).

Theorem 2 *For any positive integer k we have* CN(RW$_k$) $\leq k$.

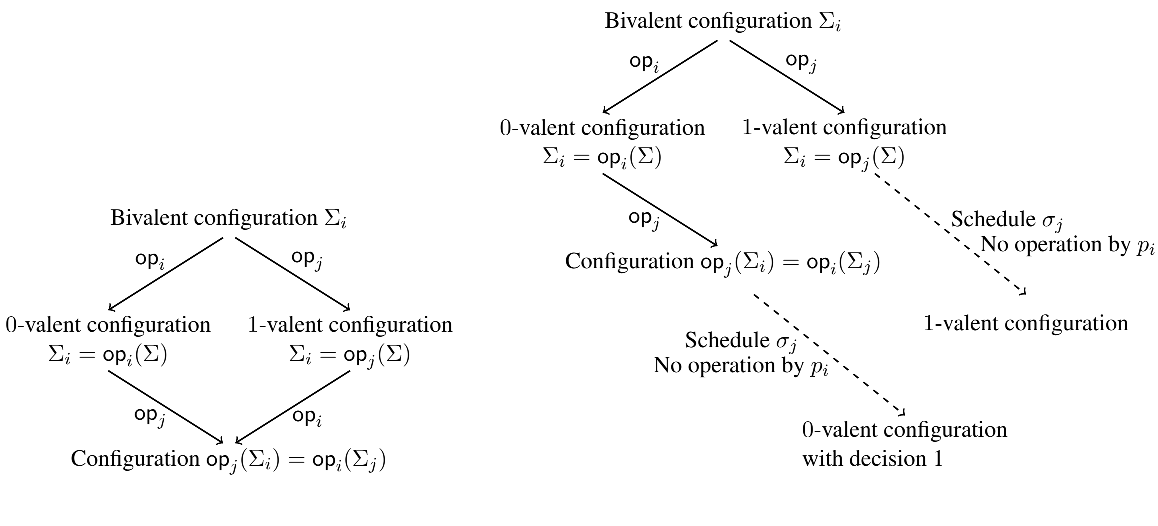

Figure 1: Schedule illustrations

[3]Assume p_i proposes 0, while p_j proposes 1. It follows from the consensus Validity property that, if all the processes except p_i crash initially, only 0 can be decided. Similarly, if all the processes except p_j crash initially, only 1 can be decided. It follows that the corresponding initial configuration is bivalent.

The proof can be seen as a straightforward generalization of the proof given in [12], which shows that atomic registers (i.e., RW_1 registers) have consensus number 1.

Proof As in [5], starting with an algorithm A assumed to implement consensus, and an initial bivalent configuration, the proof consists in building an execution of A in which there is no maximal schedule. Consequently, all its configurations are bivalent, from which follows that the schedule is infinite: A does not satisfy the consensus Termination property.

Hence, let us consider a read/write wait-free system of $(k+1)$ processes, enriched with any number of RW_k objects. As A is assumed to terminate, each of its executions generates a maximal schedule, i.e., produces a bivalent configuration Σ after which there is no more bivalent configurations. The proof is a classical case analysis depending on whether the next operation issued by each process is a read or write operation, and whether they are on the same or different RW_k objects. Let p_i and p_j be two processes whose next operations to execute in Σ are op_i and op_j, producing the 0-valent configuration $\Sigma_i = \mathsf{op}_i(\Sigma)$, and the 1-valent configuration $\Sigma_j = \mathsf{op}_j(\Sigma)$, respectively.

- Case 1 (same as Lemma 1 in [5], left size of Figure 1): The operations op_i and op_j are on different RW_k objects. We have then $\mathsf{op}_j(\mathsf{op}_i(\Sigma)) = \mathsf{op}_i(\mathsf{op}_j(\Sigma))$ (being on different objects, the operations commute without side effect), from which we conclude that this configuration is bivalent, which contradicts the fact that Σ is maximal.
- Case 2: The next operations op_i and op_j issued by p_i and p_j are on the same RW_k object and one of them (e.g., op_i) is a read. In this case, there is a schedule σ_j, starting from the 1-valent configuration $\Sigma_j = \mathsf{op}_j(\Sigma)$, in which all the processes except p_i (which stops for an arbitrary long period or crashes) issue operations and eventually decide. As $\Sigma_j = \mathsf{op}_j(\Sigma)$ is 1-valent, they decide 1.

  Let us now consider $\mathsf{op}_j(\Sigma_i) = \mathsf{op}_j(\mathsf{op}_i(\Sigma))$. This configuration differs from $\Sigma_j = \mathsf{op}_j(\Sigma)$ only in the local state of p_i (which read the RW_k object in the configuration $\mathsf{op}_j(\Sigma_i) = \mathsf{op}_j(\mathsf{op}_i(\Sigma))$, while it does not in $\Sigma_j = \mathsf{op}_j(\Sigma)$) See an illustration on the right size of Figure 1. Let us apply the schedule σ_j to configuration $\mathsf{op}_j(\Sigma_i) = \mathsf{op}_j(\mathsf{op}_i(\Sigma))$. This is possible because no process (except p_i) can distinguish $\mathsf{op}_j(\mathsf{op}_i(\Sigma))$ from $\mathsf{op}_j(\Sigma)$. From the schedule σ_j, it follows that p_j decides 1, contradicting the fact that the configuration $\Sigma_i = \mathsf{op}_i(\Sigma)$ is 0-valent.
- Case 3: In Σ, the next operation by each process is a write, and these write operations are on the same RW_k object $KREG$[4]. The reasoning is similar to Case 2. Let $\Sigma_i = \mathsf{op}_i(\Sigma)$ be 0-valent, and $\Sigma_j = \mathsf{op}_j(\Sigma)$ be 1-valent. Let σ_j be a schedule, starting from Σ_j in which
  - (a) the first $(k-1)$ operations are the write of $KREG$ invoked by the $(k-1)$ processes different from p_i and p_j.
  - (b) all processes, except p_i, execute steps until each of them decides, and
  - (b) p_i executes no operation.

  Let us notice that such a schedule is possible because, in Σ, the next operation of each process is a write into $KREG$ (Case assumption, which implies item (a)[5]), and the algorithm A terminates (hence each correct process invokes the consensus operation and decides, which implies item (b)).

  Let $\mathsf{op}_j\sigma_j$ denote the schedule composed of op_j followed by σ_j. As $\Sigma_j = \mathsf{op}_j(\Sigma)$ is 1-valent, all processes involved in $\mathsf{op}_j\sigma_j$ (i.e., all processes except p_i) decide 1.

  Let us now consider the monovalent state Σ_i, in which p_j applies op_j. Let us observe that no process, except p_i, can distinguish Σ_j from $\mathsf{op}_j(\Sigma_i)$ (they have the same local states in both). It

[4]The intuition that underlies this case is the following. While p_i can be the first process that writes a value (say 0) in $KREG$ (thereby producing a 0-valent configuration) and then pauses for an arbitrarily long period, it is possible that the next process writes 1, and the $(k-1)$ other processes write also a value, whose net effect is the elimination of the value written by p_i from the current window.

[5]The important point is here the following: in σ_j no process different from p_i can know the value written in $KREG$ by p_i.

follows that the schedule $\mathsf{op}_j\sigma_j$ (executed previously from Σ) can also be executed from Σ_i. The first k operations of this schedule are a write operation on $KREG$ issued by each process different from p_i. Moreover, at the end of this schedule, all the processes (except p_i, which is not involved in $\mathsf{op}_j\sigma_j$) decide 1. This contradicts the fact that Σ_i is 0-valent, which concludes the proof. $\Box_{Theorem\ 2}$

5 Conclusion

This paper first introduced a new type of concurrent object, parameterized by a positive integer k, namely an atomic read/write sequence which can be accessed by a read and a write operation. Each write adds a new value at the end of the sequence, while a read returns the last k written values. This generic object, called k-sliding read/write register, has an instance for each positive integer k. The instance $k = 1$ corresponds to the classical atomic read/write register, which is the most basic object of computing science [20]. Then, the paper has shown that the consensus number of such a k-parameterized object is k. Hence, this object family covers the whole spectrum of Herlihy's consensus hierarchy, a noteworthy pedagogical property. From a technical point of view, this result may help better understand the synchronization power of concurrent objects. Moreover, it is sufficient to show that an object can be implemented with a k-sliding read/write register to prove its consensus number is at most k.

Acknowledgments

The authors want to thank the referees for their constructive comments. This work has been partially supported by the French ANR project DESCARTES devoted to layered and modular structures in distributed computing.

References

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

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

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

[4] Ellen F., Gelashvili R., Shavit N., and Zhu L., A complexity-based hierarchy for multiprocessor synchronization. *Proc. 35th ACM Int'l Symposium on Principles of Distributed Computing (PODC'16)*, ACM Press, pp. 97-106 (2016)

[5] 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)

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

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

[8] 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)

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

[10] Kuo T.T., Kim H.E., and Ohno-Machado L., Blockchain distributed ledger technologies for biomedical and healthcare applications. *Journal of the American Medical Informatics Association*, 24(6):1211-1220 (2017)

[11] Lamport L., On interprocess communication, Part I: basic formalism. *Distributed Computing*, 1(2):77-85 (1986)

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

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

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

[15] Nakamoto S., Bitcoin: a peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf (2008)

[16] Perrin M., Spécification des objets partagés dans le systèmes répartis sans attente. *PhD Thesis*, 201 pages (2016)

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

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

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

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

[21] Wood G., Ethereum: a secure decentralised generalised transaction ledger. http:// bitcoinaffiliatelist.com/wp-content/uploads/ethereum.pdf (2014)

A The consensus number of the ledger object

Leader object A *ledger* is an atomic list-like object which provides processes with two operations denoted $\mathsf{read}()$ and $\mathsf{append}()$. When a process p_i invokes $\mathsf{append}(v)$, the pair $\langle i, v\rangle$ (also called block or record) is appended at the end of the list (actually, according to the application that uses a ledger, additional control information might be added to the pair $\langle i, v\rangle$). When a process invokes $\mathsf{rread}()$ it obtains the *whole* sequence of operation issued so far by the processes. Hence, no pair (block or record) is ever suppressed from a ledger. More developments on the ledger object can be found in [18].

One of the very first uses of a ledger object was in crytocurrencies, where the underlying implmementation mechanism it is called *blochchain*. A block or record can be a set transactions (as in the Bitcoin [15] or the Ethereum [21] applications), notarial deeds, medical observations [10], etc.

k-Bounded ledger object Let us consider the notion of a k-bounded ledger [18]. Such a ledger keeps only the k last values appended to the ledger. Hence, the classic ledger is an ∞-ledger. More developments on the ledger object can be found in [18].

Consensus number It is easy to see that a k-bounded ledger and a k-sliding read/write register are the same object. It follows from this observation that the consensus number of a ledger object is $+\infty$.