A Wealth of Sub-Consensus Deterministic Objects

Eli Daian
School of Computer Science, Tel-Aviv University, Israel
eliyahud@post.tau.ac.il

Giuliano Losa
Computer Science Department, University of California, Los Angeles, CA, USA
giuliano@cs.ucla.edu

Yehuda Afek
School of Computer Science, Tel-Aviv University, Israel
afek@post.tau.ac.il

Eli Gafni
Computer Science Department, University of California, Los Angeles, CA, USA
eli@ucla.edu

Abstract

The consensus hierarchy classifies shared an object according to its consensus number, which is the maximum number of processes that can solve consensus wait-free using the object. The question of whether this hierarchy is precise enough to fully characterize the synchronization power of deterministic shared objects was open until 2016, when Afek et al. showed that there is an infinite hierarchy of deterministic objects, each weaker than the next, which is strictly between i and $i+1$-processors consensus, for $i \geq 2$. For $i = 1$, the question whether there exist a deterministic object whose power is strictly between read-write and 2-processors consensus, remained open.

We resolve the question positively by exhibiting an infinite hierarchy of simple deterministic objects which are equivalent to set-consensus tasks, and thus are stronger than read-write registers, but they cannot implement consensus for two processes. Still our paper leaves a gap with open questions.

2012 ACM Subject Classification Theory of computation → Distributed computing models

Keywords and phrases shared memory, distributed algorithms, wait-free, set consensus

Digital Object Identifier 10.4230/LIPIcs.DISC.2018.17

Funding The work of Yehuda Afek and Eli Gafni was partially supported by the United States-Israel Binational Science Foundation (grant 2014226). This material is based upon work supported by the National Science Foundation under Grant No. 1655166.

1 Introduction

Shared memory objects have been classified by Herlihy [19] by their consensus number, where the consensus number of an object O is the maximum number of processes which can solve the consensus task in the wait-free model using any number of copies of O [1]. Herlihy also

[1] Read-write registers are also usually allowed, in addition to copies of O, but this is superfluous since any non-trivial object can implement bounded-use registers [7], and bounded-use suffices when solving a task.

showed that n-consensus objects are universal for n processes, meaning that, for n processes, any other object can be implemented wait-free using n-consensus objects.

Until recently, it was not known whether an object of consensus power n can be implemented wait-free using n-consensus objects (i.e., objects that can be used to solve consensus among at most n processes) in a system of more than n processes (as a special case, the Common2 [2, 5] conjecture stipulates that all objects of consensus number 2 can be implemented using consensus for 2 processes). If this were the case, then the consensus hierarchy would offer a complete characterization of the synchronization power of distributed objects.

Addressing this question, requires first to precisely define the computation model used and the notion of synchronization power. Several object binding models exists, e.g. with a notion of ports, such as in the hard-wired and soft-wired binding models [11], or without ports, such as in the oblivious model [20]. There are also several ways to compare synchronization power, such as using non-blocking implementations or wait-free implementations, and by restricting the comparison to the power to implement tasks.

In this paper, we work in the oblivious object model. Moreover, we are just concerned with the power of objects to wait-free solve task defined over finite number of processors. It is easy to see that for this if we have an implementation of the object the implementation does not need to be a wait-free implementation, it is enough that it will be non-blocking, or as called in other places lock-free.

In 2016, Afek et al. [1] constructed for every $n \geq 2$ an infinite sequence of deterministic objects (in the oblivious model) $O_{n,k}$, $k \in \mathbb{N}$, of consensus number n, and such that $O_{n,k}$ cannot be used to obtain a non-blocking implementation of $O_{n,k+1}$ in a system of $nk + n + k$ processes. Thus, for every n, the $O_{n,k}$ objects have strictly increasing synchronization power, as measured by the non-blocking implementation relation. This shows that consensus number alone is not sufficient to characterize the synchronization power of deterministic objects at levels $n \geq 2$ of the consensus hierarchy. As a special case, this also refutes the Common2 conjecture.

However, the case for consensus number 1 remained an open question, and it was conjectured that any deterministic object of consensus number 1 is equivalent to read-write registers, meaning that the object can solve exactly the same tasks that are solvable with read-write registers, no more, no less.

Herlihy [18] presented a consensus number 1 object that cannot be implemented wait-free from read-write register. But nevertheless it was implemented non-blocking (lock-free) from read-write registers, thus it did not refute the conjecture that every consensus number 1 object can be implemented non-blocking from read-write registers. Chan et al. [12] showed that for every set-consensus task, there exists an equivalent soft-wired non-deterministic object.

The main result of this paper refutes the above conjecture by presenting a deterministic object, Write and Read Next (WRN_k), in the oblivious binding model, satisfying:

▶ **Theorem 1.** *For all integers $k \geq 3$, there is a deterministic object, WRN_k, whose consensus number is 1 but which cannot be implemented non-blocking from registers in a system of $n > k$ processes.*

The second result of this paper applies to a one-shot variant, 1sWRN_k, of WRN_k. Assuming that the object may be accessed at most once by each process and that no two processes use the same argument in their invocation, we show the following theorem:

▶ **Theorem 2.** *$1sWRN_k$ and $(k, k-1)$-set consensus have equivalent synchronization power (i.e., each can implement the other).*

Since $(k, k-1)$-set-consensus is strictly weaker than $(k+1, k)$-set-consensus, this gives rise to an infinite hierarchy among the WRN$_k$ objects, such that 1sWRN$_{k'}$ objects are stronger than (can implement but not be implemented from) 1sWRN$_k$ objects if $k < k'$. Since 1sWRN$_k$ objects have more synchronization power than simple read-write registers, and cannot solve the consensus task for 2 processes, this shows the existence of an infinite number of object classes between simple read-write registers and 2-consensus.

The rest of the paper is structured as follows. The model is given in section 2. The WRN$_k$ object and its one-shot variant 1sWRN$_k$ are presented in Section 3. We show two set consensus implementations that use these objects in section 4. A construction of 1sWRN$_k$ from $(k, k-1)$-set consensus object is presented in section 5. WRN$_k$ is proved to be weaker than 2-consensus in Section 6. The implied infinite hierarchy is presented in Section 7. Finally, conclusions and open questions are discussed in section 8.

2 Model

We follow the standard asynchronous shared memory model with oblivious objects, as defined in [1], in which processes communicate with one another by applying atomic operations, called steps, to shared objects. Each object has a set of possible values or states. Each operation (together with its inputs) is a partial mapping, taking each state to a set of states. A shared object is *deterministic* if each operation takes each state to a single state and its associated response is a function of the state to which the operation is applied.

A *configuration* specifies the state of every process and the value of every shared object. An *execution* is an alternating sequence of configurations and steps, starting from an initial configuration. Processes behave in accordance with the algorithm they are executing. If C is a configuration and s is a sequence of steps, we denote by Cs the configuration (or in the case of nondeterministic objects, the set of possible configurations) when the sequence of steps s is performed starting from configuration C.

An *implementation* of a sequentially specified object O consists of a representation of O from a set of shared base objects and algorithms for each process to apply each operation supported by O. The implementation is *deterministic* if all its algorithms are deterministic. The implementation is *linearizable* if, in every execution, there is a sequential ordering of all completed operations on O and a (possibly empty) subset of the uncompleted operations on O such that:

1. If op is completed before op' begins, then op occurs before op' in this ordering.
2. The behavior of each operation in the sequence is consistent with its sequential specification (in terms of its response and its effect on shared objects).

An implementation of an object O is *wait-free* if, in every execution, each process that takes sufficiently many steps eventually completes each of its operations on O. The implementation is *non-blocking* if, starting from every configuration, if enough steps are taken, then there exists a process that completes its operation. Note that a wait-free implementation is also a non-blocking implementation. In the rest of this paper, we discuss only deterministic and linearizable wait-free implementations.

A *task* specifies what combinations of output values are allowed to be produced, given the input value of each process and the set of processes producing output values. A wait-free or non-blocking solution to a task (both notion are equivalent when consider algorithms that solve tasks) is an algorithm in which each process that takes sufficiently many steps eventually produces an output value, and such that the collection of output values satisfies the specification of the task given the input values of the process.

A task is solvable wait-free if and only if it is solvable non-blocking. This is because, in a non-blocking implementation of a bounded problem, at least one processor eventually terminates. A processor that terminates stops participating, and thus, because the implementation is non-blocking, another process eventually terminates, and so on until all processes that take sufficiently many steps have terminated, which fulfills the wait-free requirement. More generally, for any problem in which there is a bound on the number of operations that processors must complete, there is no difference between non-blocking ind wait-free.

In the *consensus task*, each process, p_i, has an input value x_i and must output a value y_i that satisfies the following two properties:

Validity. Every output is the input of some process.

Agreement. All outputs are the same.

We say that an execution of an algorithm solving consensus *decides a value* if that value is the output of some process.

The *k-set consensus task*, introduced by [14, 15], is defined in the same way, except that agreement is replaced by the following property:

k-agreement. There are at most k different output values.

Note that the 1-set consensus task is the same as the consensus task.

An object has *consensus number n* if there is a wait-free algorithm that uses only copies of this object and registers to solve consensus for n processes, but there is no such an algorithm for $n+1$ processes. An object has an infinite consensus number if there is such algorithm for each positive integer n.

For all positive integers $k < n$, an (n, k)-set consensus nondeterministic object [10] supports one operation, `propose`, which takes a single non-negative integer as input. The value of an (n, k)-set consensus object is a set of at most k values, which is initially empty, and a count of the number of `propose` operations that have been performed on it (to a maximum of n). The first `propose` operation adds its input to the set. Any other `propose` operation can nondeterministically choose to add its input to the set, provided the set has size less than k. Each of the first n `propose` operations performed on the object *nondeterministically* returns an element from the set as its output. All subsequent `propose` operations hang the system in a manner that cannot be detected by the processes.

A variant of the consensus task is the election task, in which all participating processes propose their own identifiers (rather than proposing some value). It also has the variable of k-set election task, that is basically a k-set consensus task, in which the identifiers of the processes are proposed. It was shown in [3] that the k-set consensus task is computationally equivalent to the k-set election task.

The k-strong set election task is a k-set election task, with the following self election property:

Self Election. If some process p_i decides on p_j, then p_j also decides on p_j.

It was shown in [9] that the k-strong set election task can be implemented using k-set election implementations, and thus the k-set election and k-strong set election tasks are computationally equivalent.

Algorithm 1 A sequential specification of the atomic WRN operation of a WRN$_k$ object.

```
1: function WRN($i, v$)                      ▷ $i \in \{0, \ldots, k-1\}$, $v \neq \bot$
2:     $A[i] \leftarrow v$
3:     return $A[(i+1) \mod k]$
4: end function
```

Algorithm 2 $(k-1)$-Set consensus using a WRN$_k$ object.

```
1: function Propose($v_i$)                   ▷ For process $P_i$, $0 \leq i < k$
2:     $t \leftarrow$ WRN($i, v_i$)                  ▷ $t$ is a local variable.
3:     if $t \neq \bot$ then return $t$
4:     else return $v_i$
5:     end if
6: end function
```

3 Write and Read Next Objects

For every $k \geq 2$, we introduce the WriteAndReadNext$_k$ (or WRN$_k$) object, that has a single operation – WRN. This operation accepts an index i in the range $\{0, \ldots, k-1\}$, and a value $v \neq \bot$. It returns the value v' that was passed in the previous invocation to WRN with the index $(i+1) \mod k$, or $\bot$ if there is no such previous invocation.

A possible implementation of WRN$_k$ consists of k registers, $A[0], \ldots, A[k-1]$, initially initialized to $\bot$. A sequential specification of the atomic WRN operation is presented in Algorithm 1.

The OneShotWRN$_k$ (or 1sWRN$_k$) object is similar to WRN$_k$, but any index can be used at most once. Any attempt to invoke 1sWRN with the same index twice is illegal, and hangs the system in a manner that cannot be detected by any process.

Note that the requirement that processes do not use the same argument in their invocation is reminiscent of the soft-wired model, in which there cannot be concurrency on a port. We could have chosen to specify 1sWRN$_k$ in the soft-wired binding model. This would have avoided ad-hoc assumptions about how processes use of the 1sWRN$_k$ object. We opted not to do so in order to use the oblivious object-binding model exclusively.

For $k = 2$, WRN$_2$ is simply a SWAP object, whose consensus number is known to be 2 [19]. From now on, we assume $k \geq 3$, unless stated otherwise.

4 Solving $(k, k-1)$-Set Consensus using WRN$_k$ Objects

4.1 Solution in a System of k Processes

For any $k \geq 3$, a WRN$_k$ object can solve the $(k, k-1)$-set consensus task for k processes with unique ids taken from $\{0, ..., k-1\}$, using the following algorithm (also described in Algorithm 2): Assume the processes are $P_0, \ldots, P_{k-1}$, and their values are $v_0, \ldots, v_{k-1}$. Process P_i invokes a 1sWRN with index i and value v_i. If the output of the operation, t, is $\bot$, P_i decides v_i. Otherwise, it decides t.

Since it is illegal for a process to propose multiple values (with the same ID) in the set consensus task, WRN can be replaced by 1sWRN, that is invoked at most once with each index.

▶ **Claim 3.** *Algorithm 2 is wait free.*

▶ **Claim 4.** *The first process to perform WRN decides its own proposed value.*

Proof. Since it is the first one to invoke `WRN`, the output of `WRN` is $\perp$, and hence the process decides on its own proposed value. ◀

▶ **Claim 5.** *Let P_i be the last process to perform `1sWRN`. So P_i decides the proposal of $P_{(i+1) \mod k}$.*

Proof. Since P_i is the last one to invoke `WRN`, $P_{(i+1) \mod k}$ has already completed its `WRN` invocation. Theretofore, P_i receives $v_{(i+1) \mod k}$ as the output from `WRN`. Hence, P_i decides the value of $P_{(i+1) \mod k}$. ◀

▶ **Claim 6** (Validity). *A process P_i can decide its proposed value, or the proposed value of $P_{(i+1) \mod k}$.*

▶ **Claim 7.** *A process P_i decides its own proposed value if $P_{(i+1) \mod k}$ have not invoked `WRN` yet.*

▶ **Corollary 8** ($(k-1)$-agreement). *Assume the proposals are pairwise different (there are exactly k different proposals). So at most $k-1$ values can be decided.*

Proof. Let P_i be the first process to invoke `WRN`, and P_j be the last process to invoke `WRN`. From Claim 4, P_i decides its proposal. From Claim 5, P_j decides the proposal of $P_{(j+1) \mod k}$. From claim 7, no process decides the proposal of P_j. ◀

▶ **Corollary 9.** *Algorithm 2 solves the $(k-1)$-set consensus task for k processes.*

▶ **Corollary 10.** *$1sWRN_k$ and WRN_k cannot be implemented from atomic read-write registers. Hence, $1sWRN_k$ and WRN_k are stronger than registers.*

4.2 Solution in a System with k Participating Processes Out of Many

Assuming that each process has a unique name in $\{0, \ldots, k-1\}$ might be a strong limitation in some models. In this section, we assume we have at most k participating processes, whose names are taken from $\{0, \ldots, M-1\}$, where $M \gg k$.

In [4, 6], wait-free algorithms have been shown that use registers only to rename k processes from $\{0, \ldots, M-1\}$ to k unique names in the range $\{0, \ldots, 2k-2\}$. So we shall relax our assumption, and assume now we have at most k participating processes, whose names are in $\{0, \ldots, 2k-2\}$. Let us consider the set of functions $\{0, \ldots, 2k-2\} \to \{0, \ldots, k-1\}$, call it $\mathcal{F}$. So $|\mathcal{F}| = (2k-1)^k$ is finite, and we can fix an arbitrary ordering of $\mathcal{F} = \left\{f_1, \ldots, f_{(2k-1)^k}\right\}$.

The $(k-1)$-set consensus algorithm for k processes is described in Algorithm 3. It uses $(2k-1)^k$ instances of WRN$_k$ objects, $W[1], \ldots, W\left[(2k-1)^k\right]$. First, the process name is renamed to be $j \in \{0, \ldots, 2k-2\}$. Then, for each $\ell \in \left\{1, \ldots, (2k-1)^k\right\}$ (in this exact order for all processes), the process invokes $W[\ell]$`.WRN` with the index $f_\ell(j)$, and the proposed value v_j. If the result of such a `WRN` operation returns a value different than $\perp$, the process immediately decides on this returned value, and returns without continuing to the following iterations. If the process received $\perp$ from all the `WRN` operations on $W[1], \ldots, W\left[(2k-1)^k\right]$, it decides its own proposed value.

▶ **Claim 11** (Validity). *Every decided value in Algorithm 3 was proposed by some process.*

Algorithm 3 $(k-1)$-Set consensus for k processes out of many using WRN_k objects.

```
1:  shared array of $\mathrm{WRN}_k$ objects $W[\ell]$, $1 \le \ell \le (2k-1)^k$
2:  function Propose($v$)                    ▷ For process whose name is in $\{0,\ldots,M-1\}$
3:      $j \leftarrow$ Rename                                        ▷ $j \in \{0,\ldots,2k-2\}$
4:      for $\ell = 1,\ldots,(2k-1)^k$ do
5:          $i \leftarrow f_\ell(j)$                      ▷ $i \in \{0,\ldots,k-1\}$ is a local variable.
6:          $t \leftarrow W[\ell]$.WRN($i, v$)                          ▷ $t$ is a local variable.
7:          if $t \neq \perp$ then return $t$
8:          end if
9:      end for
10:     return $v$                         ▷ Reaching here means $t$ was $\perp$ in all iterations
11: end function
```

Proof. Each process can only write its proposal to the WRN objects, and hence only proposal values or $\perp$ can be returned from the `WRN` operations. Therefore, if a `WRN` operation performed by the process P does not return $\perp$, it returns a proposal of some process Q, and hence P decides on the proposal of Q. If P gets only $\perp$ from all the `WRN` invocations, it decides on its own proposal. ◀

▶ **Claim 12.** *For every iteration number $1 \le \ell \le (2k-1)^k$, there is a process that invokes $W[\ell]$.`WRN` in Algorithm 3, and the first such process returns $\perp$.*

Proof. The first process to invoke $W[\ell]$.`WRN` returns $\perp$ by the definition of the `WRN` objects, and hence it also continues to the next iteration. Using induction, it is clear that a process gets to the first iteration and continues to the second one, and hence there is a process that accesses $W\left[(2k-1)^k\right]$, and the first such process returns $\perp$. ◀

▶ **Corollary 13.** *There is a process that invokes $W\left[(2k-1)^k\right]$.`WRN` in Algorithm 3, and the first such process decides on its proposed value.*

▶ **Claim 14.** *Assume a process P got the output $x \neq \perp$ from its invocation of $W\left[(2k-1)^k\right]$.`WRN`. x is the value of another process Q, that invoked $W\left[(2k-1)^k\right]$.`WRN` before P.*

▶ **Corollary 15.** *Assume exactly k inputs were proposed to Algorithm 3. Also assume the processes P and Q proposed the values x and y, respectively, and assume P decides on y. Q does not decide on x.*

▶ **Claim 16.** *Assume all k processes access the construction of algorithm 3, each with a different input. There is a process P that decides on the value of another process Q.*

Proof. Let R be the set of new names of the processes after renaming them in line 3, $|R| = k$. Hence there is a mapping $f_{\ell^\star} \in \mathcal{F}$ such that $\{f_{\ell^\star}(i) \mid i \in R\} = \{0, 1, \ldots, k-1\}$. Either some process returns before iteration $\ell^\star$, or all of them reach iteration $\ell^\star$.

In the former case, process P quits in iteration $\ell' < \ell^\star$, and P gets a proposal v of another process from $W[\ell']$, and decides v.

In the latter case, let j_P be the name of P after the renaming in line 3. Let P be the last process to invoke $W[\ell^\star]$.`WRN`. So P invoked it with the index $f_{\ell^\star}(j_P)$. Let Q be the process that invoked $W[\ell^\star]$.`WRN` with the index $(f_{\ell^\star}(j_P)+1) \mod k$ (there is such process because of the selection of $\ell^\star$). Q invoked $W[\ell^\star]$.`WRN` before P, and hence the P's invocation of $W[\ell^\star]$.`WRN` results in the proposal of Q. Therefore, P decides on the proposal of Q. ◀

Algorithm 4 Implementing relaxed WRN$_k$ using 1sWRN$_k$ and registers.

```
1: shared 1sWRN_k object
2: shared array of registers $A[i]$, $0 \le i < k$, initialized to 0
3: function RlxWRN($i, v$)                          ▷ $0 \le i < k$, $v \neq \bot$
4:     Inc($A[i]$)                                  ▷ Increment $A[i]$ by 1.
5:     $c \leftarrow$ Read($A[i]$)                  ▷ $c$ is a local variable.
6:     if $c = 1$ then return 1sWRN($i, v$)
7:     else return $\bot$
8:     end if
9: end function
```

▶ **Corollary 17** ($(k-1)$-agreement)**.** *Assume exactly k inputs were proposed to Algorithm 3. So there is a process P whose proposal is not decided by any process.*

Proof. Let v_i and d_i be the proposal and decision values of process P_i. Let A be the set of processes P_i such that $x_i \neq y_i$. From Claim 16, $A \neq \emptyset$.

Each process $P_i \in A$ has an iteration ℓ_i in which d_i was returned from its invocation of $W[\ell_i]$.WRN. Let ℓ' be the minimal such iteration, and let $P_i \in A$ be the last process to invoke $W[\ell']$.WRN.

No value was decided by any process in iteration $\ell < \ell'$, and hence v_i was not decided by any process in these iterations. The value v_i is unknown to $W[\ell']$ before P_i invokes $W[\ell_\prime]$.WRN. Therefore, v_i cannot be returned by any $W[\ell_\prime]$.WRN invocation prior to P_i's invocation. In P_i's invocation the value $d_i \neq v_i$ is returned. From the selection of i, every $W[\ell_\prime]$.WRN invocation after P_i's invocation returns $\bot$, and hence no process returns v_i in iteration ℓ'.

P_i have not participated in any latter iteration, and hence v_i was not seen by any WRN object in such an iteration, so it could not be returned from any WRN invocation. Therefore, v_i is not returned by any process also after iteration ℓ'. ◀

▶ **Corollary 18.** *Algorithm 3 solves the $(k-1)$-set consensus task for k processes whose names are taken from $\{0, \ldots, M-1\}$.*

Algorithm 3 uses WRN$_k$ objects that cannot be trivially replaced by 1sWRN$_k$ objects, since after the renaming, processes P and Q get the new names $0 \le i < j < 2k-1$, and there is a mapping $f_\ell \in \mathcal{F}$ such that $f_\ell(i) = f_\ell(j)$. If both P and Q get to iteration ℓ, both invoke $W[\ell]$.WRN with the index $f_\ell(i) = f_\ell(j)$.

Although this fact might pose a problem, the correctness of the algorithm is based on the existence of an iteration $\ell^\star$ such that $f_{\ell^\star}$ maps all the renamed process names onto $\{0, 1, \ldots, k-1\}$. This fact is being used in the proof of Claim 16 in order to show that there is a process that decides on the proposal of another process, and hence the $(k-1)$-*agreement* property is achieved.

A relaxed implementation of WRN$_k$ using 1sWRN$_k$ is enough for implementing Algorithm 3. This relaxed implementation is described in Algorithm 4. The 1sWRN$_k$ object is protected by a counter for every legal index. This counter is a simple atomic register that can be incremented and read (each operation is a single step). When a process comes with the index i, it first increments the counter of index i, and then reads the value of that counter. If the read value is exactly 1, it is safe for the process to invoke 1sWRN (in a similar manner to the flag principle [21]). Otherwise, the process cannot tell whether it is safe to invoke 1sWRN or not, so it gives up, and returns $\bot$ directly.

▶ **Claim 19** (Safety). *At most one process invokes `1sWRN` with an index $0 \le i < k$ in Algorithm 4.*

Proof. `1sWRN` is invoked with an index i only by a process that read the value 1 (exactly) from $A[i]$. By contradiction, assume both P and Q read 1 from $A[i]$, and without loss of generality, let Q be the last process to increment $A[i]$. Since $A[i]$ is initialized to 0 and Q is not the first process to increment it, Q must have read at least 2. ◀

▶ **Corollary 20.** *Algorithm 4 is using the 1sWRN$_k$ object legally.*

▶ **Claim 21.** *If exactly k processes arrive with k different indices, `1sWRN` is invoked by every participating process in Algorithm 4.*

Proof. Every process that comes with an index i is the only one that increments $A[i]$, so it is the only one to read the value 1 from $A[i]$, and hence it will invoke `1sWRN`. ◀

Algorithm 4 of a relaxed WRN$_k$ object can be used as a substitution for the WRN$_k$ objects in algorithm 3; lines 1 and 6 should be replaced by the following lines:

```
1: shared array of WRN$_k$ objects $W[\ell]$, $1 \le \ell \le (2k-1)^k$
6:         $t \leftarrow W[\ell]$.RlxWRN$(i, v)$                    ▷ $t$ is a local variable.
```

If at round ℓ two different processes access $W[\ell]$ with the same index i, with the relaxed WRN$_k$ the underlying `1sWRN` operation might not even get invoked, in which case both processes get $\perp$ from their `RlxWRN` invocation, if a process accesses later $W[\ell]$.`RlxWRN` with the index $(i-1) \mod k$, this process might get $\perp$ and continue to the next iteration, which is the opposite of the expected behavior with regular WRN$_k$ objects.

However, in the proof of Claim 16, iteration $\ell^\star$ still exists, in which all k participating processes invoke $W[\ell]$.`RlxWRN` with a different index, and claim 21 guarantees that in iteration $\ell^\star$, the underlying 1sWRN$_k$ object gets accesses just like the regular WRN$_k$ object. Hence Algorithm 3 solves the $(k-1)$-set consensus task for k processes using 1sWRN$_k$ objects as well.

5 Constructing 1sWRN$_k$ from $(k, k-1)$-Set Consensus Implementation

In this section we present an implementation of 1sWRN$_k$ object that uses $(k, k-1)$-strong set election (i.e., if process P_i decides in the proposal of P_j, then P_j also decides on its own proposal), which can be implemented using $(k, k-1)$-set consensus [9], and registers.

The base of the implementation is an array of registers, in which each process publishes its value (using the index), and reads the published value of its successor (by the index) if such a value is published, or $\perp$ otherwise. Each process aims to return the read value of its successor, whether it is $\perp$ or not. However, the first linearized operation must return $\perp$, and if the processes return their read value, the following execution has no first linearized operation: All processes write together their values, and then read together the values of their successors.

In order to avoid such cases, the implementation uses a doorway register. This doorway is initially open (i.e., the register value is *opened*), and once a process enters through the doorway (i.e., reads the value *opened*), it closes the doorway (i.e., writes the value *closed*). The processes that pass through the doorway use the strong set election implementation, and return the read published value of their successor only if they do not win the strong set

Algorithm 5 Implementation of 1sWRN_k using $(k, k-1)$-Strong Set Election.

```
1: shared $(k, k-1)$-strong set election implementation $SSE$
2: shared MWMR register $Doorway$, initially opened
3: shared SWMR register array $R[i]$, $0 \le i < k$; initially $R[i] = \bot$ for every $i$
4: shared SWMR register array $O[i]$, $0 \le i < k$; initially $O[i] = \bot$ for every $i$
5: function 1sWRN($i, v$)    ▷ $i \in \{0, \dots, k-1\}$ is the index, $v \notin \{\bot, \emptyset\}$ is the value.
6:     $R[i] \leftarrow v$    ▷ $v$ is announced at the index $i$.
7:     if Read($Doorway$) $=$ $opened$ then
8:         $Doorway \leftarrow closed$
9:         if $SSE$.Invoke($i$) $= i$ then
10:            return $\bot$
11:        end if
12:    end if
13:    $SR \leftarrow$ Snapshot($R$)    ▷ $SR$ is a local array.
14:    $O[i] \leftarrow SR$
15:    $SO \leftarrow$ Snapshot($O$)    ▷ $SO$ is a local array.
16:    for $j = 0, 1, \dots, k-1$ do
17:        if $SO[j][i] = v$ and $SO[j][(i+1) \mod k] = \bot$ then
18:            return $\bot$
19:        end if
20:    end for
21:    return $SR[(i+1) \mod k]$
22: end function
```

election. If a process wins the strong set election, its `1sWRN` invocation returns $\bot$. Notice that using the strong set election without the doorway might result in a non-linearizable implementation: If a process completes its `1sWRN` invocation with the index $(i+1) \mod k$ before another process issues its invocation with the index i, the latter is is expected to return the value of the former. However, the latter invocation might win in the strong set election as well, in which case it would return $\bot$.

The described solution is not enough, since the result is non-linearizable. Consider the case in which the doorway has already been closed by an early invocation. Since the read and write operations are not atomic, the linearization might break between an invocation announces its value, and reads the value of its successor index.

For example, consider the following execution: (1) an invocation w_1 with the index 1 can announce its value. (2) an invocation w_2 with the index 2 announces its value. (3) The invocation w_1 encounters a closed doorway, reads the value of w_2 and returns it. (4) After w_1 completes, an invocation w_3 announces its value. (5) w_2 reads the announces value of w_3 and returns it. In this described execution, w_1 would be linearized after w_2, that would be linearized after w_3. But w_3 starts only after w_1 has completed.

In order to overcome this kind of problem, two snapshots are being taken. The first snapshot reads the announced values, and the second one is used for announcing the snapshot every invocation observes, in order to detect scenarios similar to the one described above. If an invocation w_i observes the value of its successor invocation $w_{(i+1) \mod k}$, but it also sees that there is another invocation w_j that saw the value of w_i, but did not see the value of $w_{(i+1) \mod k}$, so w_i knows that it has started before $w_{(i+1) \mod k}$ finishes, and w_i returns $\bot$. A pseudo code of the implementation is presented in algorithm 5.

Let e be a legal execution that contains invocations to `1sWRN`, as described in Algorithm 5. Denote by $\{w_i\}$ the invocations to `1sWRN`, such that w_i is the invocation with index i and input value v_i. Assume `1sWRN` was invoked for every index $0 \le i < k$ (otherwise, append the missing invocations at the end of the execution). We will now see that Algorithm 5 is a linearizable implementation of `1sWRN`.

▶ **Claim 22.** *w_i returns $v_{(i+1) \mod k}$ or $\perp$.*

▶ **Claim 23.** *There is an index $0 \leq i < k$ such that w_i returns $\perp$.*

Proof. The first invocation to check the doorway status (in line 7) invokes the strong set election, so the strong set election is invoked at least once. By definition, there is an invocation w_i that its strong set election invocation returns i, and then w_i returns $\perp$ from Algorithm 5. ◀

▶ **Claim 24.** *There is an index $0 \leq i < k$ such that w_i return $v_{(i+1) \mod k}$.*

Proof. When some invocation takes a snapshot in line 13, all invocations that enter the doorway have already registered their values in R: Assume w_i does not read v_j in R. When w_i takes the snapshot in line 13, the doorway is already closed, and v_j is not written in R. So w_j writes v_j to R in line 6 after the doorway is closed. So w_j does not enter through the doorway.

At least one invocation reads in line 13, because an invocation reads R if it does not enter the doorway, or loses in the strong set election. Let w_i be the last invocation to write in line 6 that also reads in line 13. Claim by contradiction that w_i returns $\perp$.

So there is a an index $0 \leq j < k$ such that w_i sees $SO[j][i] = v_i$ and it also sees $SO[j][(i+1) \mod k] = \perp$. In this case, when w_j takes a snapshot of R in line 13, it sees v_i in R, but not $v_{(i+1) \mod k}$. So v_i is written to R before $v_{(i+1) \mod k}$, and after the doorway is already closed. So $w_{(i+1) \mod k}$ writes to R after the doorway is closed, and after w_i writes to R, which is a contradiction to the selection of w_i. ◀

▶ **Lemma 25.** *If w_i returns $\perp$, then $w_{(i+1) \mod k}$ finishes after w_i starts.*

Proof. By a contradiction assume $w_{(i+1) \mod k}$ finishes before w_i starts. In this case, when w_i starts, $v_{(i+1) \mod k}$ is already written in $R[(i+1) \mod k]$ and the doorway is closed, and $O[(i+1) \mod k] \neq \perp$.

Since w_i returns $\perp$, it must be done in line 18 in iteration $0 \leq j < k$, when w_j saw v_i, but not $v_{(i+1) \mod k}$. Therefore, $w_{(i+1) \mod k}$ starts after w_i starts, that is after $w_{(i+1) \mod k}$ finishes, which is a contradiction. ◀

▶ **Lemma 26.** *If w_i returns $v_{(i+1) \mod k}$, then w_i finishes after $w_{(i+1) \mod k}$ starts.*

Proof. Assume w_i finises before $w_{(i+1) \mod k}$ starts. In this case, when w_i finishes, the value in $R[(i+1) \mod k]$ is $\perp$, so w_i returns $\perp$ either if it wins the strong set election, or if it reads it from $R[(i+1) \mod k]$. ◀

We now define a directed graph $G = (V, E)$, where $V = \{w_i \mid 0 \leq i < k\}$, and the set of edges is defined as follows:

- If w_i returns $\perp$, there is an edge from w_i to $w_{(i+1) \mod k}$.
- If w_i returns $v_{(i+1) \mod k}$, there is an edge from $w_{(i+1) \mod k}$ to w_i.

▶ **Claim 27.** *There is an edge from w_i to $w_{(i+1) \mod k}$ if and only if there is no edge from $w_{(i+1) \mod k}$ to w_i.*

▶ **Corollary 28.** *There are no directed cycles in G.*

Proof. The degree of each node in the graph is exactly 2, since the edges are between w_i and $w_{(i+1) \mod k}$. Therefore, with the combination of Claim 27, if there is a cycle in G, its length is k.

Assume there is a cycle of length k in G. Using claim 23, let w_{i_1} be a `1sWRN` invocation using Algorithm 5 that returns $\perp$. Since w_{i_1} returns $\perp$, the cycle is in increasing order, e.g., for every $0 \leq i < k$, there is an edge from w_i to $w_{(i+1) \mod k}$.

Using Claim 24, let w_{i_2} be a `1sWRN` invocation that returns $v_{(i_2+1) \mod k}$. From the construction of G, there is an edge from $w_{(i_2+1) \mod k}$ to w_{i_2}, which is a contradiction to claim 27. ◀

▶ **Corollary 29.** *There is a source and a sink in G.*

▶ **Corollary 30.** *The edges of G form a partial order.*

▶ **Lemma 31.** *Let p be an increasing indices directed path from w_i to w_j. That is:*

$$p = \left\langle w_i \to w_{(i+1) \mod k} \to w_{(i+2) \mod k} \to \cdots \to w_j \right\rangle$$

Then w_j finishes after w_i starts.

Proof. In this case, every $w \in p \setminus \{w_j\}$ returns $\perp$. We use induction on p to show that w_j finishes after every $w \in P$ starts. The base case is trivial: w_j finishes after it starts.

Inductively assume w_j finishes after $w_{(i+x+1) \mod k}$ starts. We now show that w_j finishes after $w_{(i+x) \mod k}$ starts. If $w_{(i+x) \mod k}$ enters through the doorway, it is impossible for w_j to finish before $w_{(i+x) \mod k}$ starts. Let us now consider the case in which $w_{(i+x) \mod k}$ encounters a closed doorway.

If $w_{(i+x) \mod k}$ reads $\perp$ from $R\left[(i+x+1)\right]$ in line 13, then it must have started before $w_{(i+x+1) \mod k}$ starts, which is before w_j finishes.

Consider the case in which $w_{(i+x) \mod k}$ reads $v_{(i+x+1) \mod k}$ from $R\left[(i+x+1)\right]$ in line 13. Since $w_{(i+x) \mod k}$ returns $\perp$, it must have been in line 18 in iteration $0 \leq \ell < k$ of the for loop of line 16. Therefore, w_ℓ sees $v_{(i+x) \mod k}$ but not $v_{(i+x+1) \mod k}$. Hence, $w_{(i+x) \mod k}$ writes to R in line 6 before $w_{(i+x+1) \mod k}$ does. It follows that $w_{(i+x) \mod k}$ starts before $w_{(i+x+1) \mod k}$ starts, that is before w_j finishes.

Hence $w_i \in p$ starts before w_j finishes. ◀

▶ **Lemma 32.** *Let p be a descending indices directed path from w_i to w_j. That is:*

$$p = \left\langle w_i \to w_{(i-1) \mod k} \to w_{(i-2) \mod k} \to \cdots \to w_j \right\rangle$$

Then w_j finishes after w_i starts.

Proof. In this case, every $w \in p \setminus \{w_i\}$ does not return $\perp$. We use induction on the length of p to show that w_i starts before w_j finishes. The base case is trivial: lemma 26 shows that w_i starts before w_j finishes if the length of p is 1.

Assume the length of p is greater than 1. Inductively we assume that any decreasing indices path shorter than p satisfies the lemma. Also assume by contradiction that w_j finishes before w_i starts. Therefore, w_j does not read v_i from R in line 13. Since w_j returns $v_{(j+1) \mod k}$, it has to read $v_{(j+1) \mod k}$ in R after $w_{(j+1) \mod k}$ writes it there. So there is an ℓ such that $w_\ell \in p$, and w_j reads $R\left[\ell\right] = v_\ell$ but $R\left[(\ell+1) \mod k\right] = \perp$.

If operation w_ℓ reads $O\left[j\right] \neq \perp$, w_ℓ would have to return $\perp$ in line 18. Since $w_\ell \in p$, it returns $v_{(\ell+1) \mod k}$. Therefore, w_ℓ reads $O\left[j\right] = \perp$ in line 15. So w_ℓ reads O before w_j finishes. Reading O in line 15 is the last operation in the shared memory, so w_ℓ finishes before w_j does. Since the path $\left\langle w_i \to w_{(i-1) \mod k} \to \cdots \to w_\ell \right\rangle$ is a decreasing path shorter than p, from the induction assumption, w_i starts before w_ℓ finishes, that is before w_j finishes. ◀

▶ **Corollary 33** (Transitivity). *Let p be a directed path from w_i to w_j in G. So w_j finishes after w_i starts.*

We build a total order of $\{w_i \mid 0 \le i < k\}$ inductively. For the base case, denote: $S^0 = \emptyset$, $T^0 = \{w_i \mid 0 \le i < k\}$.

Given S^j and $T^j \neq \emptyset$, $0 \le j < k$, we build S^{j+1} and T^{j+1} using the following construction: denote by $\tilde{T}_j$ the set of invocations $t \in T^j$, such that t has no incoming edges in G from another invocation in T^j. Since $T^j \neq \emptyset$ then also $\tilde{T}^j \neq \emptyset$, because there are no cycles in G. Let w^j be the first invocation in $\tilde{T}^j$ to perform the write in line 6 (that is, to starts running). We define S^{j+1} and T^{j+1} as follows: $S^{j+1} = S^j \cup \{w^j\}$ and $T^{j+1} = T^j \setminus \{w^j\}$. Since $\left|T^{j+1}\right| = \left|T^j\right| + 1$, this construction is well defined for $0 \le j < k$.

We define the total order $\preceq$ as follows: $w^i \preceq w^j$ if $i \le j$.

▶ **Lemma 34.** *For every $0 \le j \le k$, there are no edges from T^j to S^j.*

Proof. We use induction on j for the proof. The base case is trivial, since $S^0 = \emptyset$.

Assume there are no edges from T^j to S^j. Since $w^j \in \tilde{T}^j$, there are no edges to w^j from T^j (and there is also no edge from w^j to itself). So there are no edges from $T^{j+1} = T^j \setminus \left\{w^j\right\}$ to $S^{j+1} = S^j \cup \left\{w^j\right\}$. ◀

▶ **Corollary 35.** *w^0 returns $\perp$.*

Proof. Assume w^0 does not return $\perp$. Following the construction of G, there is an incoming edge to w^0. From lemma 34, there are no incoming edges to $S^1 = \left\{w^0\right\}$, in a contradiction. ◀

▶ **Corollary 36.** *w_i returns $\perp$ if and only if $w_i \preceq w_{(i+1) \bmod k}$.*

Proof. Assume w_i returns $\perp$. Assume $w_i = w^j$. So $w_i \in T^j$, but $w_i \in S^{j+1}$. Since w_i returns $\perp$, following the construction of G, there is an edge from w_i to $w_{(i+1) \bmod k}$. Assuming that $w_{(i+1) \bmod k} \in S^j$ would contradict lemma 34, so $w_{(i+1) \bmod k} \in T^j$, and therefore also $w_{(i+1) \bmod k} \in T^{j+1}$. So $w_i \preceq w_{(i+1) \bmod k}$. ◀

▶ **Corollary 37.** *$\preceq$ is a linearization of `1sWRN`. Therefore, algorithm 5 is a linearizable implementation of 1sWRN$_k$.*

Corollary 37 shows that 1sWRN$_k$ can be implemented using a $(k, k-1)$-set consensus implementation. This implies that 1sWRN$_k$ is equivalent to $(k, k-1)$-set consensus. In particular, 1sWRN$_k$ cannot solve the 2-process consensus task where $k \ge 3$.

6 WRN$_k$ is Weaker than 2-Consensus

Section 5 describes a linearizable construction of 1sWRN$_k$ using an implementation for $(k, k-1)$-set consensus. In this section we prove that neither WRN$_k$ objects can solve the 2-process consensus task for $k \ge 3$, using a critical-state argument [17, 19].

We follow the standard definitions of *bivalent configuration*, *v-univalent configuration* and *critical configuration*, as defined in [17, 19].

▶ **Lemma 38.** *For each $k \ge 3$, there is no wait-free algorithm for solving the consensus task with 2 processes using only registers and WRN$_k$ objects.*

Proof. Assume such an algorithm exists. Consider the possible executions of the processes P and Q of this algorithm, while proposing 0 and 1, respectively. Let C be a critical configuration of this run. Denote the next steps of P and Q from C as s_P and s_Q, respectively. Without loss of generality, we assume that Cs_P is a 0-univalent configuration, and Cs_Q is a 1-univalent configuration.

Following [19], s_P and s_Q both invoke a `WRN` operation on the same WRN$_k$.

Algorithm 6 m-set consensus for n processes using WRN$_k$ objects.

```
shared array $W[j]$ of WRN$_k$ objects, $0 \le j < \lceil \frac{n}{k} \rceil$
function Propose($v_i$)                                   ▷ For process $P_i$, $0 \le i < n$
    $t \leftarrow W[\lfloor \frac{i}{k} \rfloor]$.WRN($i \mod k, v_i$)        ▷ $t$ is a local variable.
    if $t \ne \perp$ then return $t$
    else return $v_i$
    end if
end function
```

Case 1. Both s_P and s_Q perform `WRN` with the same index i.
The configurations Cs_P and Cs_Qs_P are indistinguishable for a solo run of P, but a solo run of P from Cs_P decides 0, while an identical solo run of P from Cs_Qs_P decides 1. This is a contradiction.

Case 2. s_P and s_Q perform `WRN` with different indices, i_P and i_Q, respectively.
Since $k \ge 3$, either $i_P \ne i_Q + 1 \mod k$ or $i_Q \ne i_P + 1 \mod k$. Without loss of generality, assume that $i_Q \ne i_P + 1 \mod k$. So the configurations Cs_Ps_Q and Cs_Qs_P are indistinguishable for a solo run of P. However, the identical solo runs of P from the configurations Cs_Ps_Q and Cs_Qs_P decide 0 and 1, respectively, which is a contradiction.

Both cases resulted in a contradiction, and therefore no such algorithm exists. ◀

7 Implications

7.1 Set Consensus Ratio

A trivial implication of Section 4 is that WRN$_k$ objects can solve the m-set consensus task for n processes as long as $\frac{k-1}{k} \le \frac{m}{n}$ is satisfied. For instance, WRN$_3$ objects can be used for implementing $(12, 8)$-set consensus.

Algorithm 6 describes an implementation of the m-set consensus task for n processes using WRN$_k$ objects. It uses an array W of $\lceil \frac{n}{k} \rceil$ shared WRN$_k$ objects, where the process named i, $0 \le i < n$ invokes the `WRN` operation of $W[\lfloor \frac{i}{k} \rfloor]$ with its proposal and the index i mod k. If $\perp$ is returned, the process decides on its own proposal. Otherwise, it decides on the returned value of the invocation.

Note that Algorithm 6 can be implemented using 1sWRN$_k$ objects instead of the WRN$_k$ objects, since every index is accesses at most once.

▶ **Lemma 39.** *For every $0 \le j < \lceil \frac{n}{k} \rceil$, the set of processes $\mathcal{P} = \{P_i \mid j \cdot k \le i < (j+1) \cdot k\}$ solves the $(k-1)$-set consensus task using algorithm 6.*

Proof. This algorithm is similar to Algorithm 2, and since $|\mathcal{P}| \le k$, corollary 9 shows algorithm 6 solves the $(k-1)$-set consensus task for $\mathcal{P}$. ◀

▶ **Corollary 40.** *Algorithm 6 solves the m-set consensus task for n processes.*

7.2 Infinite Hierarchy

The combination of the results of Sections 4 and 5 imply that 1sWRN$_k$ objects have the same computational power as $(k, k-1)$-set consensus objects, e.g. 1sWRN$_k$ objects are computationally equivalent to $(k, k-1)$-set consensus objects.

The following relationship among set consensus objects is known [1, 16]:

▶ **Theorem 41.** *Let $n > k$ and $m > j$ be positive integers. Then there is a wait-free implementation of an (n, k)-set consensus object from (m, j)-set consensus objects and registers in a system of n or more processes if and only if $k \geq j$, $\frac{n}{k} \leq \frac{m}{j}$, and either $k \geq j \cdot \lceil \frac{n}{m} \rceil$ or $k \geq j \cdot \lfloor \frac{n}{m} \rfloor + n - m \cdot \lfloor \frac{n}{m} \rfloor$.*

▶ **Corollary 42** (Hierarchy of 1sWRN objects)**.** *Let $k < k'$ be two positive integers. So:*

1. *1sWRN$_k$ cannot be implemented using 1sWRN$_{k'}$ objects and registers.*
2. *1sWRN$_{k'}$ can be implemented using 1sWRN$_k$ objects and registers.*

This corollary forms an infinite hierarchy among the 1sWRN objects, such that 1sWRN$_{k'}$ objects are considered to have more computational power than 1sWRN$_k$ objects if $k < k'$. Since 1sWRN objects have more computational power than simple read-write registers, and cannot solve the consensus task for 2 processes, this hierarchy shows the existence of an infinite number of computational power classes between simple read-write registers and 2-consensus.

8 Conclusion

This paper advances our understanding of classification of deterministic shared objects. It was an open question whether there are deterministic objects that are stronger than registers, and yet incapable of solving the consensus task for two processes.

The answer to this question for nondeterministic objects is well known [18]. For the deterministic case, only recently [1] it has been shown that the consensus task alone is not enough for classifying the computational power of deterministic objects. It is suggested that the set consensus task gives a more fine grained granularity for deterministic objects power classification, however the layer of objects under 2-consensus was not discussed.

Our construction shows that set-consensus gives a more fine grained granularity in understanding the computational power of objects, even between atomic read/write registers and 2-consensus. Not only we show the existence of objects between both computational classes, we also provide an infinite hierarchy of computational classes between the two classes, defined by the set-consensus task, using the implications of [8, 9].

Even though we have a better understanding of the behavior of deterministic objects under 2-consensus, our research leaves some open questions. We have shown that for every k, there is a deterministic object that can solve the $(k, k-1)$-set consensus task. This result is extended to the (n, m)-set consensus task, where $\frac{m}{n} \geq \frac{k}{k-1} \geq \frac{2}{3}$. We do not show the existence of deterministic objects that can solve the (n, m)-set consensus task where $\frac{n}{k} < \frac{2}{3}$ without solving the 2-consensus task. More precisely, this paper does not show (or refutes) the existence of a deterministic object that can solve the 2-set consensus task for any number of processes, but is unable to solve the 2-consensus task. These questions remain open.

Finally, although the Consensus Hierarchy is not precise enough to characterize the synchronization power of objects, we may conjecture that a hierarchy based on set-consensus may be precise enough. Chan et al. [13] give an example in which set-consensus powers is not enough to characterize the ability of a deterministic object to solve the n-SLC problem. However, by definition, the n-SLC problem is not a problem in the wait-free model. Thus the conjecture that set-consensus is enough to characterize the synchronization power of deterministic shared objects in the wait-free model (in particular, their power to solve tasks wait-free) is still open.

References

1 Yehuda Afek, Faith Ellen, and Eli Gafni. Deterministic objects: Life beyond consensus. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC '16, pages 97–106, New York, NY, USA, 2016. ACM. doi:10.1145/2933057.2933116.

2 Yehuda Afek, Eli Gafni, and Adam Morrison. Common2 extended to stacks and unbounded concurrency. *Distributed Computing*, 20(4):239–252, Nov 2007. doi:10.1007/s00446-007-0023-3.

3 Yehuda Afek, Eli Gafni, Sergio Rajsbaum, Michel Raynal, and Corentin Travers. Simultaneous consensus tasks: A tighter characterization of set-consensus. In Soma Chaudhuri, Samir R. Das, Himadri S. Paul, and Srikanta Tirthapura, editors, *Distributed Computing and Networking*, pages 331–341, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

4 Yehuda Afek and Michael Merritt. Fast, wait-free (2k-1)-renaming. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '99, pages 105–112, New York, NY, USA, 1999. ACM. doi:10.1145/301308.301338.

5 Yehuda Afek, Eytan Weisberger, and Hanan Weisman. A completeness theorem for a class of synchronization objects. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing*, PODC '93, pages 159–170, New York, NY, USA, 1993. ACM. doi:10.1145/164051.164071.

6 Hagit Attiya and Arie Fouren. Adaptive wait-free algorithms for lattice agreement and renaming (extended abstract). In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '98, pages 277–286, New York, NY, USA, 1998. ACM. doi:10.1145/277697.277749.

7 Rida A. Bazzi, Gil Neiger, and Gary L. Peterson. On the use of registers in achieving wait-free consensus. *Distributed Computing*, 10(3):117–127, Mar 1997. doi:10.1007/s004460050029.

8 Elizabeth Borowsky. *Capturing the Power of Resiliency and Set Consensus in Distributed Systems.* PhD thesis, University of California in Los Angeles, Los Angeles, CA, USA, 1995. UMI Order No. GAX96-10429.

9 Elizabeth Borowsky and Eli Gafni. Generalized flp impossibility result for t-resilient asynchronous computations. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, STOC '93, pages 91–100, New York, NY, USA, 1993. ACM. doi:10.1145/167088.167119.

10 Elizabeth Borowsky and Eli Gafni. *The implication of the Borowsky-Gafni simulation on the set-consensus hierarchy.* UCLA Computer Science Department, 1993.

11 Elizabeth Borowsky, Eli Gafni, and Yehuda Afek. Consensus power makes (some) sense! (extended abstract). In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '94, pages 363–372, New York, NY, USA, 1994. ACM. doi:10.1145/197917.198126.

12 David Yu Cheng Chan, Vassos Hadzilacos, and Sam Toueg. On the number of objects with distinct power and the linearizability of set agreement objects. In *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, pages 12:1–12:14, 2017. doi:10.4230/LIPIcs.DISC.2017.12.

13 David Yu Cheng Chan, Vassos Hadzilacos, and Sam Toueg. On the classification of deterministic objects via set agreement power. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, PODC '18, pages 71–80, New York, NY, USA, 2018. ACM. doi:10.1145/3212734.3212775.

14 Soma Chaudhuri. Agreement is harder than consensus: Set consensus problems in totally asynchronous systems. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, PODC '90, pages 311–324, New York, NY, USA, 1990. ACM. doi:10.1145/93385.93431.

15 Soma Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132–158, 1993.

16 Soma Chaudhuri and Paul Reiners. Understanding the set consensus partial order using the borowsky-gafni simulation. In Özalp Babaoğlu and Keith Marzullo, editors, *Distributed Algorithms*, pages 362–379, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

17 Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

18 Maurice Herlihy. Impossibility results for asynchronous pram (extended abstract). In *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '91, pages 327–336, New York, NY, USA, 1991. ACM. `doi:10.1145/113379.113409`.

19 Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13:124–149, January 1991. `doi:10.1145/114005.102808`.

20 Prasad Jayanti. Wait-free computing. In Jean-Michel Hélary and Michel Raynal, editors, *Distributed Algorithms*, pages 19–50, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

21 Leslie Lamport. Solved problems, unsolved problems and non-problems in concurrency. *SIGOPS Oper. Syst. Rev.*, 19(4):34–44, 1985. `doi:10.1145/858336.858339`.