Deterministic Life In
1-Consensus

Dissertation submitted in partial fulfillment of the requirements for the degree of Master of Science by
Eli Daian

This work was carried out under the supervision of Professor Yehuda Afek

September 2018
Deterministic Life In
1-Consensus

Dissertation submitted in partial fulfillment
of the requirements for the degree of
Master of Science
by
Eli Daian

This work was carried out under the supervision of
Professor Yehuda Afek

September 2018
Abstract

The consensus hierarchy classifies a shared 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 exists 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.
Acknowledgments

I am deeply grateful to my advisor, Professor Yehuda Afek for his insightful comments, suggestions and warm encouragement. I would also like to thank Professor Eli Gafni and Dr. Giuliano Losa, who have provided countless ideas and counterexamples related to this thesis; it is not without our close collaboration that this work could have been carried out the way that it has.

Finally, I would like to thank my family: my parents, Becky and Arie, without whom I wouldn’t be writing these words, and who have constantly pushed me to excel, and my wife Dafna for her consistent support, encouragement and good advice during the accomplishment of this work and in general.
## Contents

1 Introduction ........................................ 1  
   1.1 Related Work ..................................... 3  

2 Model ............................................. 7

3 WRN ............................................... 11

4 Solving \((k, k - 1)\)-Set Consensus using \(\text{WRN}_k\) Objects .......... 13  
   4.1 Solution in a System of \(k\) Processes ............ 13  
   4.2 Solution in a System with \(k\) Participating Processes Out of Many ... 15

5 Constructing \(1\text{sWRN}_k\) from \((k, k - 1)\)-Set Consensus Implementation ... 21

6 \(\text{WRN}_k\) is Weaker than 2-Consensus ............... 29

7 Implications ....................................... 33  
   7.1 Set Consensus Ratio .............................. 33  
   7.2 Infinite Hierarchy ............................... 34

8 Conclusions ...................................... 35
Chapter 1

Introduction

Shared memory objects have been classified by Herlihy [20] 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 $O$. Herlihy also 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 [3, 6] 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 exist, 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 [21].

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.
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 a 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. [2] 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 [19] presented a nondeterministic consensus number 1 object that cannot be implemented wait-free from read-write registers. 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. [13] showed that for every set-consensus task, there exists an equivalent soft-wired nondeterministic 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
Theorem 1. For all integers \( k \geq 3 \), there is a deterministic object, \( \text{WRN}_k \), 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, \( \text{1sWRN}_k \), of \( \text{WRN}_k \). 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. \( \text{1sWRN}_k \) 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 \( \text{WRN}_k \) objects, such that \( \text{1sWRN}_{k'} \) objects are stronger than (can implement but not be implemented from) \( \text{1sWRN}_k \) objects if \( k < k' \). Since \( \text{1sWRN}_k \) 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 chapter 2. The \( \text{WRN}_k \) object and its one-shot variant \( \text{1sWRN}_k \) are presented in chapter 3. We show two set consensus implementations that use these objects in chapter 4. A construction of \( \text{1sWRN}_k \) objects from \((k, k-1)\)-set consensus object is presented in chapter 5. \( \text{WRN}_k \) is proved to be weaker than 2-consensus in chapter 6. The implied infinite hierarchy is presented in chapter 7. Finally, conclusions and open questions are discussed in chapter 8.

1.1 Related Work

The foundations for the classification of the synchronization power of shared objects was laid down by Herlihy in [20], where he shows that the \( n \)-consensus
task is universal for systems of $n$ processes, which gave the motivation to use it as a mainstream hierarchy for classifying the synchronization power.

Herlihy asked whether every object of consensus number 2 has a deterministic implementation from Fetch&Add. This question was later extended into asking whether all deterministic shared objects whose consensus number is exactly 2 have the same computational synchronization power (which is known as the Common2 conjecture). This is a special case of the consensus hierarchy conjecture, which claims that for every $n \geq 2$, any object of consensus number at most $n$ has a deterministic, wait-free implementation from $n$-consensus objects and registers in a system with any finite number of processes.

It has been shown [3, 6] that Fetch&Add, Test&Set, SWAP and stack objects have deterministic, wait-free implementations from 2-consensus objects and registers in a system of $n$ processes, for all positive integers $n$.

For the nondeterministic case, [19] shows a nondeterministic object of consensus number 1 that cannot be implemented from read-write registers. Furthermore, Rachman [23] defined a family of nondeterministic objects, one of consensus number $m$ for every positive integer $m$, and sketched a proof that they cannot be deterministically be implemented from registers in a system of 3 processes or from $n$-consensus objects and registers in a system of $2n + 1$ processes for any integer $n \geq 2$. In particular, there is a nondeterministic object of consensus number 2 that cannot be deterministically be implemented from 2-consensus objects and registers in a system of 5 processes.

Both these counterexamples refute the Common2 and consensus hierarchy consensus. Since Fetch&Add objects can deterministically be implemented from 2-consensus objects and registers in a system of 5 processes, this nondeterministic object cannot be deterministically be implemented from Fetch&Add objects and registers in a system of 5, the answers to Herlihy’s original question is no.

Recently, in 2016, an infinite hierarchy of deterministic objects was shown in [2] inside each layer of the consensus hierarchy, for every $n \geq 2$. The suggested objects inside each layer are deterministic set-consensus objects, that form a
hierarchy using [17]. It has been suggested that the computational power of deterministic objects is uniquely defined by the set-agreement power vector, in which the $k$’th element is the maximal amount of processes $n_k$ such that there is a deterministic, wait-free implementation using the object and registers of $k$-set consensus for $n_k$ processes.

In a more recent work [12, 14], the set consensus power vector has been investigated. It was shown that this vector is useful only for deterministic objects, by providing a counterexample of two objects, a deterministic object and a nondeterministic object, with the exact same set consensus power vector, but the nondeterministic object being computationally stronger than the deterministic one. In other words, the nondeterministic object cannot be implemented from the deterministic object.

Up to this point, the deterministic case of consensus number 1 remained an open question. This work, which is based on the initial work shown in [1], focuses on this case and shows that the registers are not alone.
Chapter 2

Model

We follow the standard asynchronous shared memory model with oblivious objects, as defined in [2], 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 notions are equivalent when considering 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 and 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. *Binary consensus* is the restriction of the consensus task in which each input value $x_i \in \{0, 1\}$.

The *$k$-set consensus task*, introduced by [16], 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 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 [4] 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 $p_i$.

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.
Chapter 3

WRN

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 \( \{0, 1, \ldots, k - 1\} \), 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, initially initialized to \( \bot \). Call them \( A[0], A[1], \ldots, A[k-1] \). A sequential specification of the atomic WRN operation is presented in Algorithm 3.1.

**Algorithm 3.1** A sequential specification of the atomic WRN operation of a WRN\(_k\) object.

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

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 [20]. From now on, we assume $k \geq 3$, unless stated otherwise.
Chapter 4

Solving \( (k, k - 1) \)-Set Consensus using \( \text{WRN}_k \) Objects

4.1 Solution in a System of \( k \) Processes

For any \( k \geq 3 \), a \( \text{WRN}_k \) object can solve the \( (k, k - 1) \)-set consensus task for \( k \) processes with unique IDs taken from \( \{0, 1, \ldots, k - 1\} \), using the following algorithm (also described in Algorithm 4.1): Assume the processes are \( P_0, P_1, \ldots, P_{k-1} \), and their values are \( v_0, v_1, \ldots, v_{k-1} \). Process \( P_i \) invokes a \( \text{WRN} \) with index \( i \) and value \( v_i \). If the output of the operation, \( t \), is \( \bot \), \( P_i \) decides \( v_i \). Otherwise, it decides \( t \).

\[\text{Algorithm 4.1} \text{ \((k - 1)\)-Set consensus using a \( \text{WRN}_k \) object.}\]

\begin{align*}
1: \text{function Propose}(v_i) & \triangleright \text{For process } P_i, 0 \leq i < k \\
2: \quad t \leftarrow \text{WRN}(i, v_i) & \triangleright t \text{ is a local variable.} \\
3: \quad \text{if } t \neq \bot \text{ then return } t \\
4: \quad \text{else return } v_i \\
5: \quad \text{end if} \\
6: \quad \text{end function}
\end{align*}
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 4.1 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 ⊥, and hence the process decides on its own proposed value. □

Claim 5. Let $P_i$ be the last process to perform WRN. 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. Therefore, $P_i$ 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 $P_{(i+1) \mod k}$ has 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 4.1 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 $\{0, 1, \ldots, k - 1\}$ 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 $\{0, 1, \ldots, M - 1\}$, where $M \gg k$.

In [5], wait-free algorithms have been shown that use registers only to rename exactly $k$ processes from $\{0, 1, \ldots, M - 1\}$ to $k$ unique names in the range $\{0, 1, \ldots, 2k - 2\}$. So we shall relax our assumption, and assume now we have at most $k$ participating processes, whose names are in $\{0, 1, \ldots, 2k - 2\}$.

Let us consider the set of functions $\{0, 1, \ldots, 2k - 2\} \to \{0, 1, k - 1\}$, call it $F$. So $|F| = (2k - 1)^k$ is finite, and we can fix an arbitrary ordering of $F = \{f_1, f_2, \ldots, f_{(2k-1)^k}\}$.

The $(k - 1)$-set consensus algorithm for $k$ processes out of many is described in Algorithm 4.2. It uses $(2k - 1)^k$ instances of WRN$_k$ objects, denoted by $W[1], W[2], \ldots, W[(2k - 1)^k]$. First, the process name is renamed to be $j \in \{0, 1, \ldots, 2k - 2\}$. Then, for each $\ell \in \{1, 2, \ldots, (2k - 1)^k\}$ (in this exact order for all processes), the process invokes $W[\ell].\text{WRN}$ 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 $W[1], W[2], \ldots, W[(2k - 1)^k]$, it decides its own proposed value.

Claim 11 (Validity). Every decided value in Algorithm 4.2 was proposed by some process.

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
Algorithm 4.2 (k − 1)-Set consensus for k processes out of many using WRN_k objects.

1: shared array of WRN_k objects \( W[\ell], 1 \leq \ell \leq (2k-1)^k \)

2: function Propose(v) \( \triangleright \) For process whose name is in \{0, 1, \ldots, M-1\}

3: \( j \leftarrow \text{Rename} \) \( \triangleright \) \( j \in \{0, 1, \ldots, 2k-2\} \)

4: for \( \ell = 1, 2, \ldots, (2k-1)^k \) do

5: \( i \leftarrow f_\ell (j) \) \( \triangleright \) \( i \in \{0, 1, \ldots, k-1\} \) is a local variable.

6: \( t \leftarrow W[\ell] . \text{WRN}(i, v) \) \( \triangleright \) \( t \) is a local variable.

7: if \( t \neq \bot \) then return \( t \)

8: end if

9: end for

10: return \( v \) \( \triangleright \) Reaching here means \( t \) was \( \bot \) in all iterations.

11: end function

proof of some process \( Q \), and hence \( P \) decides on the proposal of \( Q \). If \( P \) gets only \( \bot \) from all the \( \text{WRN} \) invocations, it decides on its own proposal. \( \square \)

\textbf{Claim 12.} For every iteration number \( 1 \leq \ell \leq (2k-1)^k \), there is a process that invokes \( W[\ell] . \text{WRN} \) in Algorithm 4.2, and the first such process returns \( \bot \).

\textbf{Proof.} The first process to invoke \( W[\ell] . \text{WRN} \) returns \( \bot \) by the definition of the \( \text{WRN} \) 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 \( W[(2k-1)^k] . \text{WRN} \), and the first such process returns \( \bot \). \( \square \)

\textbf{Corollary 13.} There is a process that invokes \( W[(2k-1)^k] . \text{WRN} \) in Algorithm 4.2, and the first such process decides on its proposed value.

\textbf{Claim 14.} Assume a process \( P \) gets the output \( x \neq \bot \) from its invocation of \( W[(2k-1)^k] . \text{WRN} \). In this case, \( x \) is the value of another process \( Q \), that invoked \( W[(2k-1)^k] . \text{WRN} \) before \( P \) did.

\textbf{Corollary 15.} Assume exactly \( k \) inputs were proposed to Algorithm 4.2. 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 4.2, 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 \( f_{\ell^*} \in \mathcal{F} \) such that \( \{f_{\ell^*}(i) \mid i \in R\} = \{0, 1, \ldots, k-1\} \). Either some process returns before iteration \( \ell^* \), or all of them reach iteration \( \ell^* \).

In the former case, process \( P \) quits in iteration \( \ell' < \ell^* \), and \( P \) gets a proposal \( v \) of another process from \( W[\ell'] \). 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 \( W[\ell^*] \). Let \( Q \) be the process that invoked \( W[\ell^*] \) before \( P \), and hence the \( P \)'s invocation of \( W[\ell^*] \) results in the proposal of \( Q \). Therefore, \( P \) decides on the proposal of \( Q \).

Corollary 17 ((\( k-1 \))-agreement). Assume exactly \( k \) inputs were proposed to Algorithm 4.2. 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 \( \ell_i \) in which \( d_i \) was returned from its invocation of \( W[\ell_i] \). Let \( \ell' \) be the minimal such iteration, and let \( P_i \in A \) be the last process to invoke \( W[\ell'] \).

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 \( W[\ell_i] \). Therefore, \( v_i \) cannot be returned by any \( W[\ell_i] \) 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 \( W[\ell_i] \) invocation after \( P_i \)'s invocation returns \( \bot \), and hence no process returns \( v_i \) in iteration \( \ell' \).
$P_i$ 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 $\ell'$.

**Corollary 18.** Algorithm 4.2 solves the $(k-1)$-set consensus task for $k$ processes whose names are taken from $\{0, 1, \ldots, M-1\}$.

The WRN$_k$ objects in Algorithm 4.2 cannot be trivially replaced by 1sWRN$_k$ objects, since after the renaming stage, processes $P$ and $Q$ get the new names $0 \leq i < j < 2k-1$, 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 $\ell$, both invoke $W[\ell]$.WRN with the same 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 $\ell^*$ such that $f_{\ell^*}$ 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 4.2. This relaxed implementation is described in Algorithm 4.3. 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 [22]). Otherwise, the process cannot tell whether it is safe to invoke 1sWRN or not, so it gives up, and returns $\perp$ directly.

**Claim 19 (Safety).** At most one process invokes 1sWRN with an index $0 \leq i < k$ in Algorithm 4.3.

**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 $A[i]$,
Algorithm 4.3 Implementing relaxed WRN\(_k\) using 1sWRN\(_k\) and registers.

1: shared 1sWRN\(_k\) object
2: shared array of registers \(A[i]\), \(0 \leq i < k\), initialized to 0
3: function \(\text{RlxWRN}(i, v)\) \(\triangleright 0 \leq i < k, v \neq \perp\)
4: \(\text{Inc}(A[i])\) \(\triangleright\) Increment \(A[i]\) by 1.
5: \(c \leftarrow \text{Read}(A[i])\) \(\triangleright c\) is a local variable.
6: if \(c = 1\) then return 1sWRN\((i, v)\)
7: else return \(\perp\)
8: end if
9: end function

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.

\textbf{Corollary 20.} Algorithm 4.3 is using the 1sWRN\(_k\) object legally.

\textbf{Claim 21.} If exactly \(k\) processes arrive with \(k\) different indices, 1sWRN is invoked by every participating process in Algorithm 4.3.

\textit{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 4.3 of a relaxed WRN\(_k\) object can be used as a substitution for the WRN\(_k\) objects in Algorithm 4.2; lines 1 and 6 should be replaced by the following lines:

1: shared array of relaxed WRN\(_k\) objects \(W[\ell]\), \(1 \leq \ell \leq (2k - 1)^k\)
6: \(t \leftarrow W[\ell] \cdot \text{RlxWRN}(i, v)\) \(\triangleright t\) is a local variable.

If at round \(\ell\) 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 \(W[\ell] \cdot \text{RlxWRN}\) 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 $\ell^*$ still exists, in which all $k$ participating processes invoke $W[\ell^*].R1xWRN$ with a different index, and Claim 21 guarantees that in iteration $\ell^*$, the underlying $1sWRN_k$ object gets accesses just like the regular $WRN_k$ object. Hence Algorithm 4.2 solves the $(k-1)$-set consensus task for $k$ processes using $1sWRN_k$ objects as well.
Chapter 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 on 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 ⊥ otherwise. Each process aims to return the read value of its successor, whether it is ⊥ or not. However, the first linearized operation must return ⊥, 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 (e.g., the register value is opened), and once
a process enters through the doorway (e.g., reads the value *opened*), it closes the doorway (e.g., 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 election. If a process wins the strong set election, its 1sWRN invocation returns ⊥. 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 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 ⊥.

The described solution is not enough though, since the result is not 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 announces 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 ⊥.
A pseudo code of the implementation is presented in Algorithm 5.1.

**Algorithm 5.1** Implementation of 1sWRN

1: shared \((k, k-1)\)-strong set election implementation \(SSE\)
2: shared MWMR register \(Doorway\), initially opened
3: shared SWMR register array \(R[i], 0 \leq i < k\); initially \(R[i] = \bot\) for every \(i\)
4: shared SWMR register array \(O[i], 0 \leq i < k\); initially \(O[i] = \bot\) for every \(i\)
5: function \(1s\text{WRN}(i, v)\) is the index, \(v \in \{\bot, \emptyset\}\) is the value.
6: \(R[i] \leftarrow v\) is announced at the index \(i\).
7: if \(\text{Read}(Doorway) = \text{opened}\) then
8: \(Doorway \leftarrow \text{closed}\)
9: if \(SSE.\text{Invoke}(i) = i\) then
10: return \(\bot\)
11: end if
12: end if
13: \(SR \leftarrow \text{Snapshot}(R)\) is a local array.
14: \(O[i] \leftarrow SR\)
15: \(SO \leftarrow \text{Snapshot}(O)\) is a local array.
16: for \(j = 0, 1, \ldots, k-1\) 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

Let \(e\) be a legal execution that contains invocations to \(1s\text{WRN}\), as described in Algorithm 5.1. Denote by \(\{w_i\}\) the invocations to \(1s\text{WRN}\), such that \(w_i\) is the invocation with index \(i\) and input value \(v_i\). Assume \(1s\text{WRN}\) was invoked for every index \(0 \leq i < k\) (otherwise, append the missing invocations at the end of the execution). We will now see that Algorithm 5.1 is a linearizable implementation of \(1s\text{WRN}\).

**Claim 22.** \(w_i\) returns \(v_i \mod k\) or \(\bot\).

**Claim 23.** There is an index \(0 \leq i < k\) such that \(w_i\) returns \(\bot\).

**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...
\[ i, \text{ and then } w_i \text{ returns } \bot \text{ from Algorithm 5.1.} \]

**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 \(\bot\).

So there is a an index \(0 < j < k\) such that \(w_i\) sees \(SO[j][i] = v_i\) and it also sees \(SO[j][(i + 1) \mod k] = \bot\). 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 \(\bot\), 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 \(O[(i + 1) \mod k] \neq \bot\).

Since \(w_i\) returns \(\bot\), 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 \( w_i \) finishes 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 25, let \( w_{i_1} \) be a 1\text{swRN} invocation using Algorithm 5.1 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 23, let \( w_{i_2} \) be a 1\text{swRN} 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:

\[ p = \langle w_i \rightarrow w_{(i+1) \mod k} \rightarrow w_{(i+2) \mod k} \rightarrow \cdots \rightarrow w_j \rangle \]

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 \( R[(i+x+1)] \) 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 the value \( v_{(i+x+1)} \mod k \) from \( R[(i+x+1)] \) 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, \( w_{\ell} \) 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:

\[
p = (w_i \rightarrow w_{(i-1)} \mod k \rightarrow w_{(i-2)} \mod k \rightarrow \cdots \rightarrow w_j)
\]

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 \( \ell \) such that \( w_{\ell} \in p \), and \( w_j \) reads \( R[\ell] = v_{\ell} \) but \( R[(\ell+1) \mod k] = \perp \).
If operation $w_\ell$ reads $O[j] \neq \bot$, $w_\ell$ would have to return $\bot$ in line 18. Since $w_\ell \in p$, it returns $v_{(\ell+1) \mod k}$. Therefore, $w_\ell$ reads $O[j] = \bot$ in line 15. So $w_\ell$ reads $O$ before $w_j$ finishes. Reading $O$ in line 15 is the last operation in the shared memory, so $w_\ell$ finishes before $w_j$ does. Since the path $\langle w_i \rightarrow w_{(i-1) \mod k} \rightarrow \cdots \rightarrow w_\ell \rangle$ is a decreasing path shorter than $p$, from the induction assumption, $w_i$ starts before $w_\ell$ 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 $\{w_i \mid 0 \leq i < k\}$ inductively. For the base case, denote:

$S^0 = \emptyset$, $T^0 = \{w_i \mid 0 \leq i < k\}$.

Given $S^j$ and $T^j \neq \emptyset$, $0 \leq j < k$, 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:

$S^{j+1} = S^j \cup \{w^j\}$

$T^{j+1} = T^j \setminus \{w^j\}$

Since $|T^{j+1}| = |T^j| + 1$, this construction is well defined for $0 \leq j < k$.

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

**Lemma 34.** For every $0 \leq j \leq k$, 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 $T^{j+1} = T^j \setminus \{w^j\}$ to $S^{j+1} = S^j \cup \{w^j\}$.

**Corollary 35.** $w^0$ returns $\bot$. 27
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 \( S^1 = \{ w^0 \} \), in a contradiction. \( \square \)

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

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

Thus \( w_i \preceq w_{(i+1) \mod k} \). \( \square \)

**Corollary 37.** \( \preceq \) is a linearization of \( 1sWRN \). Therefore, Algorithm 5.1 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 \( k \geq 3 \).
Chapter 6

WRN\(_k\) is Weaker than 2-Consensus

Chapter 5 describes a linearizable construction of 1sWRN\(_k\) using an implementation for \((k, k - 1)\)-set consensus, which shows that 1sWRN\(_k\) objects cannot be used alone with registers for solving the consensus task for 2 processes. In this chapter we prove that neither WRN\(_k\) objects can solve the 2-process consensus task for \(k \geq 3\), using a critical state argument [18, 20].

**Definition 38** (\(n\)-Valent Configuration). Given an algorithm for solving the consensus task, a configuration \(C\) of this algorithm is \(n\)-valent if in every execution whose prefix is \(C\) there are exactly \(n\) different decided values.

**Definition 39** (Bivalent Configuration). A bivalent configuration is a 2-valent configuration.

**Definition 40** (Univalent Configuration). An univalent configuration is a 1-valent configuration.

**Definition 41** (\(v\)-Univalent Configuration). A \(v\)-univalent configuration is a univalent configuration \(C\) such that every execution whose prefix is \(C\) decides \(v\).

**Definition 42** (Critical Configuration). A critical configuration is a bivalent
configuration $C$, such that any possible step $s$ from it results in an univalent configuration $Cs$.

Lemmas 43 and 44 stand in the basic of every standard critical state argument [18, 20].

**Lemma 43.** In a wait free algorithm for solving the consensus task for two processes $P$ and $Q$ proposing 0 and 1, respectively, the initial configuration is bivalent.

*Proof.* If $P$ runs alone from the initial configuration, it has to decide 0, leaving the system in a 0-univalent configuration. Similarly, if $Q$ runs alone from the initial configuration, it has to decide 1, leaving the system in a 1-univalent configuration. Since no other processes run the algorithm, no other value could be possibly decided by any execution. Therefore, the initial configuration is bivalent.

**Lemma 44.** A wait free algorithm for solving the consensus task for two processes $P$ and $Q$ has a critical configuration when $P$ proposes 0 and $Q$ proposes 1.

*Proof.* Since the algorithm is wait-free, the length of an execution is bounded by a finite number, and hence there is a finite amount of executions.

Since the initial configuration is bivalent (from Lemma 43) and is a prefix of 0-univalent configuration and a 1-univalent configuration, and there is a finite number of executions (and hence configurations), there must be a critical configuration for this algorithm and these inputs.

**Lemma 45.** For each $k \geq 3$, 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 [20], $s_P$ and $s_Q$ both invoke a WRN operation on the same WRN$_k$.

Case 1. Both $s_P$ and $s_Q$ perform WRN with the same index $i$.

The configurations $C_{s_P}$ and $C_{s_Q}s_P$ are indistinguishable for a solo run of $P$, but a solo run of $P$ from $C_{s_P}$ decides 0, while an identical solo run of $P$ from $C_{s_Q}s_P$ 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 $k \geq 3$, either $i_P \neq i_Q + 1 \mod k$ or $i_Q \neq i_P + 1 \mod k$.

Without loss of generality, assume that $i_Q \neq i_P + 1 \mod k$ (the other case is symmetric). So the configurations $C_{s_P}s_Q$ and $C_{s_Q}s_P$ are indistinguishable for a solo run of $P$. However, the identical solo runs of $P$ from the configurations $C_{s_P}s_Q$ and $C_{s_Q}s_P$ decide 0 and 1, respectively, which is a contradiction.

Both cases resulted in a contradiction, and therefore no such algorithm exists. \qed
Chapter 7

Implications

7.1 Set Consensus Ratio

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

Algorithm 7.1 describes an implementation of the \(m\)-set consensus task for \(n\) processes using WRN\(_k\) objects. It uses an array \(W\) of \(\left\lceil \frac{n}{k} \right\rceil\) shared WRN\(_k\) objects, where the process named \(i\), \(0 \leq i < n\), invokes the WRN operation of \(W\left[\left\lfloor \frac{i}{k} \right\rfloor\right]\) with its proposal and the index \(i \mod k\). If \(\bot\) is returned, the process decides on its own proposal. Otherwise, it decides on the returned value of the invocation.

\[\text{Algorithm 7.1} \quad \text{m-set consensus for n processes using WRN}_k \text{ objects.}\]

\begin{verbatim}
1: shared array \(W[j]\) of WRN\(_k\) objects, \(0 \leq j < \left\lceil \frac{n}{k} \right\rceil\)
2: function Propose\((v_i)\) \(\triangleright\) For process \(P_i\), \(0 \leq i < n\)
3: \(t \leftarrow W\left[\left\lfloor \frac{i}{k} \right\rfloor\right]\).WRN\((i \mod k, v_i)\) \(\triangleright t\) is a local variable.
4: \(\text{if } t \neq \bot \text{ then return } t\)
5: \(\text{else return } v_i\)
6: \(\text{end if}\)
7: \(\text{end function}\)
\end{verbatim}

Note that Algorithm 7.1 can be implemented using 1sWRN\(_k\) objects instead of the WRN\(_k\) objects, since every index is accessed at most once.

33
**Lemma 46.** The set of processes $\mathcal{P} = \{P_i \mid j \cdot k \leq i < (j + 1) \cdot k\}$ solves the $(k - 1)$-set consensus task using Algorithm 7.1 for every $0 \leq j < \left\lceil \frac{n}{k} \right\rceil$.

*Proof.* This algorithm is similar to Algorithm 4.1, and since $|\mathcal{P}| \leq k$, Corollary 9 shows Algorithm 7.1 solves the $(k - 1)$-set consensus task for $\mathcal{P}$. 

**Corollary 47.** Algorithm 7.1 solves the $m$-set consensus task for $n$ processes.

### 7.2 Infinite Hierarchy

The combination of the results of chapters 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 [2, 17]:

**Theorem 48.** 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 $k \geq j \cdot \left\lceil \frac{n}{m} \right\rceil$ or $k \geq j \cdot \left\lfloor \frac{n}{m} \right\rfloor + n - m \cdot \left\lfloor \frac{n}{m} \right\rfloor$.

**Corollary 49** (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.
Chapter 8

Conclusions

This work 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 [19]. For the deterministic case, only recently [2] 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 [9, 8].

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 $\frac{n}{m} \geq \frac{k}{k-1} \geq \frac{2}{3}$. We do not show the existence of deterministic objects that can solve the $(n, m)$-set consensus task where $\frac{n}{n} < \frac{2}{3}$ 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. [15] 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.
Bibliography


בעבדה זו אנו מפריכים את ההשערה, באמצעות הצגה של אובייקט דטרמיניסטי פשוט, שрешה את בעיית הסכמת הקבוצה, ולכל חוק ישרה, אך לא כל חלינו מцитים פיתון ש与此 בנמצא מועבר. בעבדה זו אנו מציגים היררכיה אינסופית של אובייקטים כאלו, שבכל אחד מהם חזק יותר מאוגר, אךInvocation לא יכול להביא להסכמה בין שני מעבדים.
The consensus problem between 𝑛 processors is universal in a system with 𝑛 processors. Hence, given a number of 𝑛 processors, it is possible to implement a Wait-Free solution for every object to a system of 𝑛 processors through the solution of the consensus problem for 𝑛 processors. Any shared object shared between an object and a pair of processors can be defined as the consensus number (Consensus Number), which is the maximum number of processors for which a Wait-Free solution for the consensus problem exists in terms of using a shared object and atomic read and write operations only.

If there is a solution to the consensus problem using a shared object and read and write pairs for every number of processors, it is said that the consensus number of the object is infinity. In fact, there is a hierarchy of objects: if one object is stronger than another, then its consensus number is greater than that of the other.

To this date, it was common to classify the computational power of shared objects using their consensus number. Additionally, the question was raised: are objects with the same consensus number equivalent in terms of computational power? In other words, is this hierarchy complete, or are there sub-hierarchies within each level of the hierarchy?

Only recently, in 2016, a hierarchy of objects was constructed within each layer of the hierarchy of consensus numbers, where the consensus number is at least 2. These infinite hierarchies use deterministic objects that solve the problem of set consensus (in Hebrew: Set Consensus), which is not a deterministic problem, in order to evaluate the computational power of deterministic objects, and in fact, found "holes" in the classic hierarchy.

The question about deterministic objects that cannot solve the consensus problem (i.e., their consensus number is 1, like registers) remained open. Can there be a deterministic object that is stronger than a register (i.e., does not have a Wait-Free solution in terms of only read and write operations), but still cannot achieve consensus between 2 processors? The prevailing hypothesis to date is that all deterministic objects that cannot achieve consensus between 2 processors are equivalent in terms of synchronization power.

The question of whether an object can be stronger than a register (i.e., there is a Wait-Free solution for 2 objects) and at the same time still cannot achieve a consensus between 2 processors remains open. This is the interesting question that drives the research on the topic.
The basic example of shared objects is or registers (Read Write Registers), which allow one to perform two operations: writing a value into the register and reading the last value written to the register. We can classify registers into several categories:

- How many processors can read from the register? Is it just one processor, or multiple processors?
- How many processors can write to the register? Is it just one processor, or multiple processors?
- How robust is the order of execution? That is, what happens if a processor performs a read while another processor is performing a write?

The motivation behind these classifications is to understand if there are registers that are stronger than other registers, that is, if there is a certain type of registers that cannot be implemented using a different type of registers. Surprisingly, the answer to all these questions is no. That is, no matter what the answer is to each one of these questions, there exists a construction of Wait-Free registers that can be used among processors.

At the beginning of the 1980s, the Consensus (Consensus) problem was presented. In this problem, each processor comes with its own input (a proposal), and all processors are required to reach a shared agreement on one of the proposals. It is easy to solve this problem using a mutual exclusion mechanism, but there is no solution for Wait-Free to the problem for 2 or more processors using Read-Write registers, and therefore we need to use stronger objects that cannot be implemented using a Read-Write registers alone.

However, one can solve the problem Wait-Free for 2 processors using more complex objects such as stacks (Stack) or queues (Queue). But these objects do not allow solving Wait-Free for 3 or more processors. Similarly, there is an object for each natural number \( n \), which allows solving the Consensus problem for \( n \) processors, but not for \( n+1 \) processors.

There are also objects that allow solving Wait-Free for any number of processors, such as atomic compares and swaps (Compare and Swap) operations.
埸 cần

عالم החישוב המבוזר עוסק בין היתר במודלים חישוביים שונים, בהם משתתפים מספר מעבדים, שיש להם משימה משותפת которую הם נדרשים להשלים. המודל החישובי המבוקש הוא מודל הזיכרון המשותף. במודל זה, כל מעבד מתקשר אחד עם השני באמצעות ביצוע פעולות על האובייקטים המשותפים בכל המעבדים (ולכן חיים בזיכרון המשותף), אך כל מעבד רשאי גם לבצע פעולות מקומיות, שאינן נראות למעבדים אחרים, לפחות עד שתוצאה של פעולות אלה משפיעה על אובייקטים בזיכרון המשותף.

בעשחיבת החדש מבוזר אсинכרוני, נהוג להניח שהמעבדים המשתתפים לא בהכרח משתתפים בזמן אותו זמן, ושהקצב החישוב וההתקדמות שלהם משתנה בין מעבד למעבד. כמו כן, מעבד מסוים עשוי להיעצר לפרק זמן מסוים ולא לבצע צעדים בזיכרון המשותף. לכן, נהוג לסווג אלגוריתמים מבוזרים לקטגוריות של בעיות שעשויות להיגרם כתוצאה מהנחות אלה.

למשל, אלגוריתמים מבוזרים רבים כוללים מנגנון של מניעה הדדית (בלעז Mutual Exclusion, הידוע גם בקיצור Mutex). במנגנון זה, קטעים קריטיים באלגוריתם, שאותם רשאי לבצע רק מעבד אחד במקביל, לא מועברים ל他の מעבדים בשתייה בקודם ale תказанו. אם מעבד מפסיק לבצע פעולות בזמן שהוא בקטע הקריטי, מעבדים אחרים נדרשים להמתין לו עד שהיע unserialize מבצע פעולות וייצא מהקטע הקריטי. דוגמאות למצבים אלה הן מצבי ה-
Dead Lock וה-
Live Lock, בהם מספר מעבדים מונעים את התקדמות האלגוריתם, גורמים לקבלי מתוכן מאל אי אפשר.

ל olacağı, נהוג לסווג אלגוריתמים מבוזרים לקטגוריות. אלגוריתם
Lock-Free הוא אלגוריתם שבו מובטח שתוך מספר סופי של פעולות בזיכרון המשותף (כותר צעדים), לפחות מעבד אחד ישיג התקדמות בדרך לפתרון. אלגוריתם Wait-Free הוא אלגוריתם ששב המקרא ש ${(M)}-כותר פעולות ביויקון המשותף (לחליק פעמים), וה שתוך מעבדاختriors התוכנית בדרכ_parוורסיויה.ยา האלגוריתם

לfram ביבית יסיים את התוכנית ולא יסיים כל שוחזרו לא בכומות.

לfram ביבית יסיים את התוכנית ולא יסיים כל שוחזרו לא בכומות.
מגוון של אובייקטים
deטרמיניסטים שלＡ＆Ｍ
לحسبים

חיבור זה הוגש כחלק מהדרישות לקבלת התואר ‏"מוסמך אוניברסיטה" – M.Sc. באוניברסיטת תל-אביב

על ידי
אליהו דיין

העבודה הוכנה בהדרכתו של
פרופסור יהודה אפק

אלול התשע"ח

הפקולטה لمデעים מדוייקים ע"ש ריימונד ובברלי סאקלר
בית הספר למדעי המחשב ע"ש בלבטניק

מגוון של אובייקטים
deטרמיניסטים שלＡ＆Ｍ
לحسبים

חיבור זה הוגש כחלק מהדרישות полученהתואר ‏"מוסמך אוניברסיטה" – M.Sc. באוניברסיטת תל-אביב

על ידי
אליהו דיין

העבודה הוכנה בהדרכתו של
פרופסור יהודה אפק

אלול התשע"ח
다면ו של אובייקטים
dטרמיניסטים שלآ מוסלמים
ל teasers

חיירז זה הגשה חלחול מדריך ל.LayoutStyle לבקת התואר
"מוסכם אנרגיסטי" – בואנהאנדסדהו תל-אביב
בין לשתי למדעי המחשב
על דידי
אליהו דני

העבדה הוכנה בהדרכה של
פרופסור יהודה אפק

אלל התשיעית