Consensus Power Makes (Some) Sense!

(Extended Abstract)

Elizabeth Borowsky[*] Eli Gafni[†] Yehuda Afek[‡]

Abstract

This paper presents partial results of an ongoing investigation into the computability power of processors that are computing wait-free while synchronizing through various deterministic linearizable synchronization objects. Our main result is that any task over $\leq 2n$ processors, solvable using objects that cannot implement $n+1$ process consensus, can be solved using only n process consensus objects. We also identify two new object implementation notions: *protocol-implementation* and *task-implementation.* We show that objects that cannot implement $n+1$ process consensus are protocol-implementable by n process consensus objects. A corollary of the latter result is that the class of objects that do not implement $n+1$ process consensus is closed under composition (i.e., in the terminology of [17], Herlihy's hierarchy is *robust*). With these two results, we substantiate for the first time Herlihy's consensus number n notion for objects with fan-in greater than n.

[*]Computer Science Department, University of California, Los Angeles, CA 90024, USA.

[†]Computer Science Department, University of California, Los Angeles, CA 90024, USA. Part of this work done while visiting ATT Bell Labs.

[‡]Computer Science Department, Tel-Aviv University, Israel 69978. Part of this work done while visiting ATT Bell Labs.

1 Introduction

Several recent papers [5, 12, 21] have examined the computability power of t-resilient computation with communication via read-write memory. In light of these results, the next natural question is how computability power increases when communication is through media more powerful than read-write memory. Although some results of this flavor appear in [5, 14, 7], they are limited because they apply to the specific instance of media, namely, n process consensus objects, used by n processors. We will address the general model of any media made of deterministic linearizable objects.

Herlihy [10] proposed characterizing objects in a hierarchy according to their ability to implement consensus. An object is of *consensus number n* if it can implement n process consensus but cannot implement $n+1$ process consensus. For this classification to make any tangible sense, the least one would expect is that the set of tasks solvable by $m > n$ processors using objects of consensus number n be no larger than the set solvable by using only n process consensus objects, and the most one would expect is that objects of the same consensus number implement each other. Can this be true?

Not in general, since it follows from [5, 7] and from theorem 3.9 (composition theorem) below that there exists a consensus number n object, namely, the composition of an n process consensus object and an object of $2n+1$ process 2-set consensus, that can solve a task (namely, $2n+1$ processes 2-set consensus) that is not solvable solely with n consensus objects.

This complication arises only when we reach $2n+1$ processors. As an interim result, we were able to show that for protocols over $m \leq 2n$ processors, all consensus number n objects have the same task-solving power. Our difficulty in going beyond $2n$ is not unlike the difficulty encountered in extending the one failure of [4] to the multiple failures in [5, 12, 21]: up to $2n$ processors, we deal with simple connectivity of the full-information view graph; above $2n$ (i.e., above $2 = 2n/n$), things are a bit more complex.

On the way to proving even this limited result, we had to resolve the question raised by Jayanti [17] and Kleinberg and Mullainathan [18] regarding the robustness of Herlihy's hierarchy, that is, whether an object that is the composition (in the sense of the Cartesian product, as explained section 2 below) of two consensus number n objects is itself a consensus number n object. That the hierarchy is indeed robust is a corollary of our composition theorem, which states that the composition of two objects each unable to solve (when used in a "canonical protocol" as explained later) $n+1$ process consensus is also an object that cannot solve $n+1$ process consensus.

We derive the composition theorem by first observing the following crucial fact: An object A (single copy) in conjunction with any number of n process consensus objects and read-write registers does not solve the $n+1$ process consensus task if and only if the view graph of any protocol for $n+1$ processors using a single copy of A together with any number of n process consensus objects and read-write registers is connected. We are then able to show (for $n+1$ processors) that we are able to replace the object in any fixed protocol by n process consensus objects. In other words, an observer controlling interleaving over the communication objects (everything but A) and the timing of invoking inputs to the object will find it impossible to determine whether we actually have object A or a set of n process consensus objects. Now, when we have two objects, we do induction on the number of steps of the protocol to show that, from the point of view of each single object, it is interacting only with communication variables of power no greater than n consensus. Thus each single object can itself be replaced by n process consensus objects up to this point in the protocol. Consequently, the whole protocol can be simulated by n process consensus objects, giving the required result that the view graph is connected. This gives us an alternative characterization of consensus number n objects – a single copy of A together with $n-1$ process consensus objects solves the n consensus task, but A together with n process consensus, does not solve the $n+1$ consensus task.

2 Model

2.1 Linearizable Deterministic Objects

Following [11], we intuitively perceive objects as "hardware black-boxes." They interface with their environment through "pins" or "ports." Various input voltages and output voltages correspond to various inputs and outputs, respectively. When using objects, one may envision a restrictive environment, in which the connection between a pin and a processor has to be hard-wired and consequently must be specified before runtime (hard environment), or a less restrictive environment, in which software may control which processor attaches to what pin at what time (soft environment); this is not unlike the question of whether an array's dimension must be predeclared or may be left to be determined at runtime. We show (lemma 3.1) that a hard-wired object implementing n process consensus implements soft-wired n process consensus as well. We make the assumption that all the consensus objects in this paper are soft-wired.

Functionally, an object is a variant of a Mealy machine [15]. In response to an input at a pin, the object makes a transition from one state to another state and responds with an output to the pin. The object is deterministic in the sense that an input applied at a pin in a particular state determines uniquely the next state and the response. We assume that all possible inputs can be applied when the object is in any state. Thus, the object is defined for a serial sequence of inputs. The object is linearizable [11] if it contains a scheduler that interfaces between the pins and the Mealy machine and applies inputs and returns responses serially. The scheduler is fair in the sense that it does not leave an input pending indefinitely. The scheduler communicates with ports through an asynchronous channel. Our use of objects is such that when a processor applies an input at a pin, the processor remains attached to the pin and is blocked from doing anything else until it receives a response. Only one processor at a time may be attached to a pin.

We prove (lemma 3.2) that if there exists a state s in object A, reachable from the initial state, such that A implements n process consensus when started at s, then A implements n process consensus when started at its initial state. Thus we will assume w.l.o.g. that every state can serve as an initial state.

To summarize the above mathematically: An object is (P, S, I, O, δ), where P is a set of ports, S is a set of states, I is a set of sets of inputs, one set I_i for each port i, O is a set of sets of outputs, one set O_i for each port i, and δ is a transition function mapping port, state, and input at the port to next state and output at the port.

Roughly, in I/O parlance, processors interact with an object via actions $invoke_i(d_i \in I_i)$, where i is a port, and $receive\_response_i(d'_i \in O_i)$, which are input and output actions, respectively, for the object. The invoke action enables the object transition corresponding to the input and the port. The internal action of the transition disables the transition and enables the output action of the response. We assume that a processor's behavior in an execution is well formed in the sense that each invoke alternates with the corresponding response with no intermediate action by the processor.

2.1.1 Object Composition

A *composition* of two objects A and B is essentially an object that is the two objects side by side. The composed object has pins, which are the collection of the pins of A and B, and state space, which is the Cartesian product of the two state spaces. Each transition in the composed object is a transition of one of the individual objects and is associated with the corresponding state component. Thus, in the specification of the composed object, only states that differ in one component may have a transition between them. One may now see why the two objects implement the composed object, and vice versa. Clearly, the composition operation is commutative and transitive.

Mathematically: If $A = (P_A, S_A, I_A, O_A, \delta_A)$ and $B = (P_B, S_B, I_B, O_B, \delta_B)$, where we assume that all elements in sets A and B are distinct, then $A \times B = (P_A \cup P_B, S_A \times S_B, I_A \cup I_B, O_A \cup O_B, \delta_{A\times B})$, where $\delta_{A\times B}$ acts in the appropriate way on the state of A or B according to which port the input is from.

2.2 Object Implementation

Essentially, the notion of implementation of object type A by B is (informally) that object type A can be "replaced by" object B. We use three degrees of strength of this notion in our paper. One notion is that A is replaced by B such that, behind the interface between the memory and A, the primitive used is object B rather than A and the replacement cannot be detected in any environment. This is the strongest notion and will be indicated by saying *implements* [10].

A weaker notion of implementation involves replacing the object undetectably within a given protocol P. Note that a protocol limits the sequence of inputs at the interface to the object. We say that B *P-implements* A if inside the given protocol we can replace A by a collection of B's and read-write registers.

We want to now refer to a collection of protocols, but we want to limit the power of the sequences at the interface so that the sequences do not correspond to an arbitrary environment. Thus, for a collection of objects CV, we say that B *(CV, m)-implements* A if, in any protocol over m processors which uses a single copy of A and objects from CV, we can implement A by a collection of B's.

In particular, we do not want the environment to be able to produce a family of sequences that cannot be produced by the object A itself. We say that a protocol over m processors is *canonical(A, n)* if it uses a single copy of A (the object) and any number of n process consensus and read-write objects (communication variables). Let $A \times C$ be the composition of A and C and assume that both are (CV, m)-implementable by B; then, as we will show, B (CV, m)-implements $A \times C$.

The weakest of our implementation notions is that tasks solvable by using A (and CV) can be solved using objects of type B (and CV) instead. We will indicate this type of implementation by saying *task-implements.*

With the first two kinds of implementations, we need to address the issue of whether the environment/protocol is terminating or nonterminating, ultimately a question of whether one can imitate the fairness of the scheduler. In the general environment case, this issue corresponds to the question of one-shot vs. long-lived. We deal only with the terminating case in this paper.

Whether our three are materially distinct – that is, whether for each pair of implementation notions there is a pair A and B such that B implements A with respect to one implementation notion but not with respect to another – is a research topic in its own right. However, examples showing that P-implements is distinct from implements are easy to construct.

Finally, as observed in [17, 18], there are many ways to interpret how one "substitutes" B for A. Having different object types, as Nir Shavit[20] pointed out, corresponds to having different models of computation, as in finite automata versus Turing machines. Having different numbers of copies of an object is analogous to investigating the power of Turing machines with more or less space. In the latter case, the model of computation is fixed but one imposes "resource bounds" on the computation; this is essentially a complexity question. Afek and Stupp's [2] investigation into the power of a single object is an example of a complexity result in this area. Characterizing the power of read-write [5, 12, 21] is a computability result; it applies to any number of read-write registers. Thus, if we want to investigate computability power, our implementation notion must refer to a collection, that is, to whether there exists a collection of type B objects together with read-write registers that can replace a given collection of type A objects (together with read-write registers). Neiger [19] has indicated that deterministic objects of consensus number $n \geq 2$ can implement read-write registers; with this result, the implementation definition becomes more streamlined.

We postpone formal definitions to the full paper, but semi-formally we assume that the reader is familiar with the standard notion of system S_1 *implementing* S_2 [1]. Let system S_2 use objects of type $A_1, A_2, ...$, let S_1 use objects of type $B_1, B_2, ...$, and let S_3 be a system that can be composed with both S_1 and S_2.

1. We say that the B class of objects implements the A class if for any S_2 system, there is an S_1 system such that for any S_3 system, S_1 composed with S_3 implements S_2 composed with S_3.

2. We say B S_3-implements A if for all S_2 systems composed with a fixed S_3, there exists an S_1 composed with S_3 that implements S_2 composed with S_3.

3. We say that B protocol-implements A if for any S_2 and S_3 systems, there exists an S_1 system such that S_1 composed with S_3 implements S_2 composed with S_3 (i.e., unlike (1) above, S_1 may depend on S_3).

4. We say that the B class of objects (CV, m)-implements the A class if B protocol-implements A when S_3 is restricted to using objects of the CV class and is a protocol over m processors.

5. Finally, we say that B m-task-implements A if any task solved by an S_2 system with m processors is solved by S_1 with m processors.

2.3 View Graphs

Let P be a wait-free protocol. A *view* v_i of processor i in an execution (or run) of P is its local state upon halting (halting state). We attach the name *view* to a halting state because in general we will refer to full-information protocols [9] and in such protocols a state is everything the processor "viewed" throughout the execution. The protocol induces a map from final views to outputs for the protocol. Out of the views of all executions of P, we build a view-graph *complex* of *simplexes*. A simplex in the complex is a set of views, one per processor, all from the same execution. Pictorially, it is a graph in which the nodes correspond to views and two views are connected by an edge if there exists a common execution to which the nodes correspond. A simplex is a complete subgraph. The *dimension* of a simplex is the number of nodes minus one, a *face* of a simplex is any subsimplex, and a *solo execution view* is one in which the processor sees only itself take steps in the protocol.

3 Characterization of Objects that Do Not Implement $n+1$ Process Consensus

In this section, we provide a characterization condition for when a deterministic object cannot solve $n+1$ processor consensus. The connectivity over all view graphs of canonical(A, n) protocols is the determining factor in the characterization. By construction, we show that if all view graphs for canonical(A, n) protocols are connected, then n process consensus can $((n\text{-consensus}, r/w), n+1)$-implement A. Alternatively, we show that if there exists a disconnected view graph for a canonical(A, n) protocol, then the object indeed implements $n+1$ process consensus. We begin by proving a few necessary lemmas.

3.1 Some Properties of Deterministic Objects

Here we show that a hard-wired n processor consensus object is equivalent to a soft-wired one. Thus, throughout the remainder of the paper, all consensus objects are assumed to be soft-wired. We also show that if a deterministic object has consensus number n, then the object cannot implement $n+1$ process consensus regardless of the state to which the object is initialized.

Lemma 3.1 *For a fixed set of processors $1, \ldots, m$, a soft-wired n process consensus object can be implemented from hard-wired n process consensus objects.*

Proof (by construction) Arrange a set of $\binom{m}{n}$ hard-wired n process consensus objects in a sequence. Let each of these be hard-wired for a unique n processor subset of the m possible processors. Processors now apply to the objects to which they are hard-wired in the order induced by the sequence. Each processor uses the output from an object as input to the next object it accesses. The final output for a processor is determined by the last object it accesses in the sequence.

To see why this works, consider the hard-wired object of the sequence which is accessed by all processors. This object acts as a filter passing a single value to all processors. This guarantees that any object accessed subsequently will have input and output fixed to this value. □

Lemma 3.2 (Initial State Lemma) *If an object A can implement n process consensus when initialized to some reachable state S, then n processors can implement consensus using copies of A initialized to the start state.*

Proof Let each processor have a private copy of object A initialized to the start state, and let the processors be ordered $1, \ldots, n$. Since state S is reachable, let each processor apply a sequence of inputs so that its copy of A reaches state S. At this point, a processor i writes into the memory 'ready-i'. A processor then takes a snapshot of memory to see which objects are ready. Following the idea of lemma 3.1, each processor executes consensus, in the order induced by the processor numbers, on the set of objects it sees ready. Again processors submit the output from one consensus as the input to the next. The first object to be pronounced ready will necessarily be accessed by all processors; thus the output value for that consensus will be the final output for each processor. □

3.2 Simplex Agreement

The primary technique used in implementation of tasks and objects is that of simplex agreement [13]. Given a complex, in our case the view graph for a protocol (G), the simplex agreement problem is for each processor to take a node of the complex as output such that all output nodes lie in a common simplex. Each simplex corresponds to a run of the protocol, and processors are required to take views labeled by their own IDs. Let S_k be a subset of processors. Then $S_k(G)$ is the subgraph of G in which only processors in S_k take steps. Then furthermore for simplex agreement on G, if only a subset S_k of the processors wakes up, the output views taken by these processors must belong to $S_k(G)$. Thus, a processor waking up alone in the simplex agreement must adopt a view that corresponds to a solo execution.

Define two simplexes in a complex be k-connected, $k > 0$, if they share a face of dimension $k-1$ or more. Define the k-run graph of a complex to be the graph in which runs (simplices) are nodes and two nodes are connected if the corresponding simplices are k-connected. We say a *complex* is k-connected if the k-run graph of the complex is connected.

Let C be a complex. Then the *linking-sphere* of a simplex $S \in C$ with respect to C is the subcomplex induced by a all the vertices of $C - S$ which lie in simplexes in C that contain S.

Lemma 3.3 *Let A be of consensus number n, G the view graph of a canonical(A, n) protocol over m, $n < m$ processors, and S_k a subset of the m processors such that $|S_k| = k > n$; then $S_k(G)$ is $k - n$ connected.*

Proof Assume the $(k-n)$-run graph (G'_k) of $S_k(G)$ is disconnected. W.l.o.g. assume we have two connected fragments $f0$ and $f1$ in G'_k. Since initially all runs of G'_k are possible, there must be a critical state of the protocol in which any operation determines the fragment of the run [8]. Color any views of the view graph shared by runs in both fragments by distinct colors. Color remaining views by 0 if they lie in runs of fragment $f0$ and 1 if they lie in runs of fragment $f1$.

Bring the protocol to the critical state. Let $n+1$ processors using n process consensus do agreement for the consensus values associated with $f0$ and $f1$ and then simulate the k processor protocol from the critical state [5]. At least $k - n$ codes will terminate. The labels of these resulting views, if they include 1 or 0, determine the fragment in which the simulation ended. Otherwise, the $k - n$ views are all labeled by distinct colors. Notice, however, that any combination of $k - n$ colors in a simplex uniquely determines a fragment. Thus, the $n + 1$ simulating processors can waitfree agree on a fragment, contradicting the assumption that A is of consensus number n. □

Lemma 3.4 *For an m processor canonical(A, n) protocol, $n < m \leq 2n$, with view graph G, and a sequence of subsets of the processors $S_{n+1} \subset S_{n+2} \subset \cdots \subset S_l$, $l \leq m$ with $|S_i| = i$, there exists a corresponding sequence of complexes $C_{n+1} \subset \cdots \subset C_m$ with $vertexes(C_{n+1}) = vertexes(S_{n+1}(G))$, $C_i \subseteq S_i(G)$, C_i is of dimension $i - 1$ and is $i - n$-connected, such that for $n < j < k \leq l$, the linking sphere of any simplex in C_j of $j-(n+1)$ dimension or less, with respect to C_k is connected.*

Sketch of Proof We use induction on l. Lemma 3.3 establishes the basis for the induction. Let q be the smallest index for which the lemma fails. There exists a simplex S in S_i for which the lemma holds, namely the simplex in which processors wake up in order, one terminating before the other wakes up. Partition S_{n+2} into two sets of vertices. Those for which S can be extended to satisfy the lemma, $f0$, and those that do not satisfy the lemma comprise the other $f1$. We now take $q = n+2$, and we defer the general case to the full paper. Obviously there are vertexes in S_{n+1} that belong to $f1$. Let $w_0 \in f0$ be adjacent to $w_1 \in f1$. Since in C_{n+2} that solves the problem on $f0$ w_0 is adjacent to all nodes in its linking sphere adding w_1 with all the 1-dimensional simplexes that connect it to vertices on the linking sphere of w_0 would have satisfy the linking sphere condition for w_1. Thus we must conclude that w_1 is not adjacent to any vertex on the linking sphere of w_0. Consequently, w_1 cannot be adjacent to a 1-dimensional simplex in C_{n+2}. Thus the existence of two vertices of C_{n+2} in a run in which S_{n+1} woke up and after at least one processor terminated then $S_{n+2} - Sn + 1$ woke up, and the existence of vertex not in C_{n+2} are two mutually exclusive outcome. Thus two outcome reveal the fragment, contradicting the impossibility of $n + 1$ consensus. □

Theorem 3.5 *m processors, $n < m \leq 2n$, can do simplex agreement on the view graph of a canonical(A, n) protocol using only n process consensus.*

Proof Let G be the view graph for a canonical(A, n) protocol. Let each processor keep a set of views F that are *fixed* (initially the empty set), and let there be predefined paths for each pair of views, participating sets and fixed set F (notions to be defined inductively below).

Upon starting simplex agreement, processors register in the memory and then use test-and-set to partition themselves into two slotted sets, A and B, of less than or equal to n processors. Sets A and B will each synchronize through n process consensus to work as a single processor. All processors try to access slots of set A first, so if $|B| > 0$ then $|A| = n$. Both A and B then take snapshots of the memory to get a participating set of processors. A and B each propose, through n process consensus, their solo view and participating set (set of processors that they observed registered in memory). Assume some embedding of the complex in some Euclidean space of high enough dimension. Processors then do ϵ-convergence [4]. Let S_k be the union of their participating sets. There is some predefined rule that by the winners in A and B and by S_k determines a sequence $S_{n+1}, ..., S_k$ such that the winners belong to S_{n+1}. They then converge along a predefined path for the starting views and S_{n+1}. Let $\epsilon = 1/4$(length of smallest edge on the path). We will refer to this sequence of steps as the convergence phase.

After converging, sets A and B each end at points (within ϵ from each other) in the embedding. Each set writes its ending point in memory and then updates F as follows. An ending point of A or B within ϵ of a view v yields $F = F \cup \{v\}$. Otherwise, the ending point lies on an edge with endpoints v_0, v_1, in which case $F = F \cup \{v_0, v_1\}$. Any processor seeing the ending point of both sets A and B takes F to be the union of the F's corresponding to each ending point. If a processor sees a view of its own label in F, it drops out of the simplex agreement with that view as output.

Processors now repeat the convergence phase, starting from views on the predefined path on the linking sphere of views of F, and converging across the path with respect to the new participating sets. By lemma 3.4, there exists such a path for convergence. Once $m - n$ nodes are fixed, no processor will belong to set B, so set A works alone, proposing (and trivially converging to) one view at each subsequent convergence phase. Thus all processors halt with nodes of a common simplex. □

3.3 Object Implementation

We start with a few preliminaries. Let a canonical(A, n) protocol be $(n + 1)$-*concurrent* if, when $< n+1$ processors access the communication objects (everything except for A), the processor accesses are totally linearized. In other words, accesses to a set of communication objects in the protocol by a single processor can be viewed as a single atomic step. For an $(n + 1)$-concurrent protocol, when $\geq n+1$ processors access the communication objects, there are no constraints on the processor linearization. Throughout this section, we assume all protocols are $n+1$-concurrent. This assumption can easily be justified by noticing that with n process consensus we can implement n-fetch-and-add that returns the sequence of the last $n - 1$ processors linearized and the set of all processors that accessed the object. It can be seen then that only when we have $n + 1$ consecutive distinct processors linearized do we lose track of the order inside the fetch-and-add. If in addition, after a processor accesses the fetch-and-add, it also writes its view in shared memory, then exactly $n + 1$ concurrency is required for processors to lose track of the linearization. Thus we get the following lemma.

Lemma 3.6 *If the view graph G for each canonical(A, n) protocol is connected, then the subgraph G' induced by views in $n+1$-concurrent runs is connected.*

Let σ be a sequence of operations to an object A, let "$\cdot$" denote concatenation of lists, and let α be a prefix of σ. Define $Next(\sigma, \alpha)$ to be the longest sequence λ such that λ contains no two operations of the same processor and $\alpha \cdot \lambda$ is a prefix of σ. If G is a view graph of a protocol, then define G/α to be the graph induced by views in runs R of G for which α is a prefix of R.

Lemma 3.7 *If $\forall$ canonical(A, n) protocols, the view graph G is connected, then for each protocol, the subgraph G/α (for given prefix α) is connected.*

Proof G/α is the subgraph of views from runs that start with prefix α. By lemma 3.2, if G/α is disconnected, we can start executing from the state reached by prefix α and achieve $n+1$ process consensus. Thus, a disconnected view graph for a canonical(A, n) protocol must exist. By contradiction, G/α is connected. □

We now show that any canonical(A, n) protocol P for $n + 1$ processors can be P-implemented by n process consensus. Thus, in $n + 1$ processor protocols, A can be ((n-consensus,r/w),$n+1$)-implemented by n-consensus. The idea of the implementation is to iterate the simplex agreement used to implement tasks, continually building a prefix α of runs of the protocol. When α is a complete linearization of a run, the simulation terminates.

Lemma 3.8 (Object Implementation) *If $\forall$ canonical(A, n) protocols over $n+1$ processors, the view graph is connected, then n process consensus ((n-consensus, r/w), $n + 1$)-implements A.*

Proof Let G be the view graph of a canonical(A, n) protocol for $n+1$ processors. For every view and edge of G, let there be a predetermined run (n-simplex) containing the view or, for the edge, containing both views of the edge. Thus the runs associated with an edge and one of its endpoints share at least one view. Let α, initially the empty string, store a prefix of a run. For each pair of views in G/α, let there be a predetermined path between the views.

Processors execute the convergence phase of simplex agreement over the G/α. Sets choose initial views for the convergence in which α is followed by operations of only the viewing processor.

After convergence, sets A and B end at points p_A and p_B, in the embedding of the view graph, along the path used for convergence. If a point is within ϵ from a view it will map to that view, and otherwise it will map to the edge it lies on. Let R_A (resp. R_B) denote the run associated with the view or edge that is mapped to from p_A (p_B).

Set A (resp. B) writes p_A (p_B) and R_A (R_B) and then reads. W.l.o.g. assume set B reads the run of set A. If set A does not read R_B, it knows the run corresponds to either the view closest to p_A or the edge p_A lies on. Thus, A can determine whether both possible runs R_B are such that $Next(\alpha, R_A) = Next(\alpha, R_B)$. To be safe, A always assumes $R_B \neq R_A$ (if this is possible) until it sees otherwise. Since B sees both R_A and R_B, it knows for certain whether $Next(\alpha, R_A) = Next(\alpha, R_B)$. Thus each set may proceed with an assumed or certain view of what R_A and R_B are.

If $Next(\alpha, R_A) = Next(\alpha, R_B)$, then processors update α to $\alpha \cdot Next(\alpha, R_A)$. If a processor has an operation in $Next(\alpha, R_A)$, it takes the corresponding object value, goes to the CV objects to continue the steps of the protocol, and then returns to convergence to linearize again in the object. Otherwise, a processor repeats the convergence phase (with updated α) in order to linearize its own operation on the object.

If $Next(\alpha, R_A) \neq Next(\alpha, R_B)$, then R_A and R_B must share at least one view. Thus let $p*$ be the next processor to linearize in the CV objects in both runs. Since the run is $n+1$-concurrent and at least one processor can not tell the difference between R_A and R_B, $p*$ is well defined. Processor $p*$ necessarily has the same object output in both runs and so may take its output, linearize in the CV objects, and return the convergence phase to linearize again in the object. When $p*$ returns to the convergence after linearizing alone in the CV objects, it accesses set B directly, leaving A open for the other processors. With α unchanged, $p*$ starts a new convergence phase, starting from the view to which its ending point was closest in the previous convergence phase and tagging the convergence with its completed CV access. If any processor obstructs $p*$'s linearization in the CV objects, the processor reveals which of R_A and R_B was chosen. In this case, $p*$ updates α accordingly and returns to the convergence phase as usual (trying for a slot of A).

When $Next(\alpha, R_A) \neq Next(\alpha, R_B)$, all processors other than $p*$ return directly to begin a new convergence phase. They will all access slots of A. Set A starts convergence at the view closest to an ending point of the previous convergence and tags the round with $p*$'s memory access. If the convergence between A and B is nontrivial (i.e., A and B start at different points), then $p*$ linearized in the CV objects successfully. After convergence with new P_A, P_B, processors can either update α or send another processor to take an object value and linearize in the CV objects.

Continuing in this manner, at least one processor gets output for the object after each convergence phase, so the processors eventually simulate all their accesses to the object and halt with output according to the protocol. Thus A is P-implemented from n process consensus for an arbitrary protocol, and the theorem holds. □

3.4 The Composition Theorem

Theorem 3.9 (Composition Theorem)
If B protocol-implements A and B also protocol-implements C, in each case w.r.t. communication objects CV, then B protocol-implements $A \times C$ w.r.t. CV.

Sketch of Proof We induct on the number of steps of the execution of the protocol. Since A and C by themselves can be implemented, and there is no notion of simultaneous steps by both, accesses to object A may be viewed as separate from access to CV objects since B can replace them, and vice versa for accesses to object C. □

Corollary 3.10 *Herlihy's consensus hierarchy is robust.*

3.5 Characterization via Canonical Protocols

Corollary 3.11 (Object Characterization) *There exists no protocol using objects A, n process consensus, and read-write memory that solves $n+1$ process consensus iff, for all canonical(A,n) protocols over $n+1$ processors, the view graph is connected.*

Sketch of Proof

($\Rightarrow$) By definition, if there is no protocol that solves $n+1$ process consensus using copies of A, n process consensus, and read-write objects, then A is of consensus number at most n. So, by lemma 3.3, a canonical(A,n) protocol has a connected view graph.

($\Leftarrow$) Conversely, by the object implementation lemma (lemma 3.8), if for all canonical(A,n) protocols, the view graph is connected, then n process consensus protocol-implements A. By [16, 7], n process consensus cannot implement $n+1$ process consensus, so there can be no canonical(A,n) protocol implementing $n+1$ process consensus. Thus, by theorem 3.9, (composition theorem), any protocol using multiple copies of A can be implemented by n process consensus. Again, [16, 7] imply that no such protocol can implement $n+1$ process consensus. □

4 Task Implementation

The next result gives the first tangible benefit to the notion of consensus number n class. It shows that for up to m processors, $m \leq 2n$, objects in the class are interchangeable with respect to solving tasks. This is a direct result of theorem 3.5 and the composition theorem (theorem 3.9).

Theorem 4.1 (The Main Theorem) *A task T for m processors, $n \leq m \leq 2n$, which can be implemented by a collection of consensus number n objects can be implemented by n process consensus.*

Proof By theorems 3.5, and 3.9, any set of consensus number n objects can be composed to form a single consensus number n object. Thus consider task T to be implemented by a canonical(A,n) protocol, where A is the composition object. Let G be the view graph for this protocol. By lemma 3.5, processors can do simplex agreement on G using only n processor consensus objects and read-write memory. By the assumed correctness of the protocol implementing T, when processors take outputs that correspond to their views, the outputs satisfy the task. □

References

[1] M. Abadi and L. Lamport, "The Existence of Refinement Mapping," *Theoretical Computer Science,* Vol. 2, No. 82, pages 253-284, 1991.

[2] Y. Afek and G. Stupp, "Synchronization Power Dependes on the Register Size," *Proc. 34th Symp. on Foundations of Computer Science,* pages 196–206, 1993.

[3] Y. Afek, E. Weisberger and H. Weisman, "A Completeness Theorem for a Class of Synchronization Objects," *Proc. 12th ACM Symp. on Principles of Distributed Computing*, pages 159–170, 1993.

[4] O. Biran, S. Moran, and S. Zaks, "A Combinatorial Characterization of the Distributed Tasks which Are Solvable in the Presence of One Faulty Processor," *Proc. 7th ACM Symp. on Principles of Distributed Computing*, pages 263–275, 1988.

[5] E. Borowsky and E. Gafni, "Generalized FLP Impossibility Result for t-Resilient Asynchronous Computations," *Proc. 25th ACM Symp. on the Theory of Computing*, pages 91–100, 1993.

[6] E. Borowsky and E. Gafni, "Immediate Atomic Snapshots and Fast Renaming," *Proc. 12th ACM Symp. on Principles of Distributed Computing*, pages 41–51, 1993.

[7] E. Borowsky and E. Gafni, "The Implication of the Borowsky-Gafni Simulation on the Set-

Consensus Hierarchy," University of California, Los Angeles, Technical Report 930021, 1993.

[8] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *Journal of the ACM*, Vol. 32, No. 2, pages 374–382, 1985.

[9] G. N. Frederickson and N. A. Lynch, "Electing a Leader in a Synchronous Ring," *Journal of ACM*, Vol. 34, No. 1, pages 98–115, 1987.

[10] M.P. Herlihy, "Impossibility and Universality Results for Wait-Free Synchronization", *Proc. 7th ACM Symp. on Principles of Distributed Computing*, pages 276–290, 1988.

[11] M.P. Herlihy and J.M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Transactions of Programming Languages and Systems,* Vol. 12, No. 3 pages 463–492, 1990.

[12] M. Herlihy and N. Shavit, "The Asynchronous Computability Theorem for t-Resilient Tasks," *Proc. 25th ACM Symp. on the Theory of Computing*, pages 111–120, 1993.

[13] M. Herlihy and N. Shavit, "A Simple Constructive Computability Theorem for Wait-free Computation," *Proc. 26th ACM Symp. on the Theory of Computing*, in press.

[14] M. Herlihy, "Wait-Free Synchronization," *ACM Transactions on Programming Languages and Systems,* Vor. 13, No. 1, pages 123–149, 1991.

[15] J. Hopcroft and J. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.

[16] P. Jayanti and S. Toueg, "Some Results on the Impossibility, Universality and Decidability of Consensus," *Proc. 6th Workshop on Distributed Algorithms,* 1992.

[17] P. Jayanti, "On the Robustness of Herlihy's Hierarchy," *Proc. 12th ACM Symp. on Principles of Distributed Computing*, pages 145–157, 1993.

[18] J. Kleinberg and S. Mullainathan, "Resource Bounds and Combinations of Consensus Objects," *Proc. 12th ACM Symp. on Principles of Distributed Computing*, pages 133–143, 1993.

[19] G. Neiger, Rump Session Talk, 12th ACM Symp. on Principles of Distributed Computing, 1993.

[20] Nir Shavit, personal communication.

[21] M. Saks and F. Zaharoglou," Wait-Free k-Set Agreement is Impossible: The Topology of Public Knowledge," *Proc. 25th ACM Symp. on the Theory of Computing*, pages 101–110, 1993.