N-Consensus is the Second Strongest Object for $N + 1$ Processes

Eli Gafni[1] and Petr Kuznetsov[2]

[1] Computer Science Department, University of California, Los Angeles
eli@ucla.edu
[2] Max Planck Institute for Software Systems
pkouznet@mpi-sws.mpg.de

Abstract. Objects like queue, swap, and test-and-set allow two processes to reach consensus, and are consequently "universal" for a system of two processes. But are there deterministic objects that do not solve 2-process consensus, and nevertheless allow two processes to solve a task that is not otherwise wait-free solvable in read-write shared memory?

The answer "no" is a simple corollary of the main result of this paper: Let A be a deterministic object such that no protocol solves consensus among $n + 1$ processes using copies of A and read-write registers. If a task T is wait-free solvable by $n + 1$ processes using read-write shared-memory and copies of A, then T is also wait-free solvable when copies of A are replaced with n-consensus objects. Thus, from the task-solvability perspective, n-consensus is the second strongest object (after $(n+1)$-consensus) in deterministic shared memory systems of $n+1$ processes, i.e., there is a distinct gap between n- and $(n + 1)$-consensus.

We derive this result by showing that any $(n + 1)$-process protocol P that uses objects A can be emulated using only n-consensus objects. The resulting emulation is non-blocking and relies on an *a priori* knowledge of P. The emulation technique is another important contribution of this paper.

1 Introduction

Consensus [6] (n-consensus object) is a fundamental abstraction that allows n processes to agree on one of their input values. Consensus is *n-universal*: every object shared by n processes can be *wait-free implemented* using n-consensus objects (and read-write registers), i.e., the object can be "replaced" with n-consensus objects, so that the external observer cannot detect the replacement [7].

But which aspects of the universality of n-process consensus remain valid in a system of $n + 1$ processes? Ideally, one would wish to show that, in a system of $n + 1$ processes, every object that does not allow for $(n + 1)$-process consensus (to be called an object of *consensus power* less than $n + 1$) can be wait-free implemented from n-consensus objects. This would imply that every problem that can be solved by $n + 1$ processes using objects of consensus power less than $n + 1$ can also be solved using n-consensus objects.

In this paper, we address an easier question: whether every deterministic object of consensus power less than $n + 1$ can be replaced with n-consensus objects, so that the

replacement is not detectable *in a given* $(n+1)$*-process protocol* P. Thus, we consider the classical *emulation* problem: given a protocol P designed for a system with objects from a set $\mathcal{C}$, how to *emulate* P in a system with objects from a set $\mathcal{C}'$.

We show that every protocol for $n + 1$ processes using read-write registers and objects A of consensus power less than $n + 1$ can be emulated using only read-write registers and n-consensus objects. Our emulation is *non-blocking* [7]: at least one active process is guaranteed to take infinitely many (emulated) steps of P. Even though the emulation does not ensure progress for *every* active process, it helps in answering the following question: is there a *decision task* T [8] for $n + 1$ processes that can be solved by objects of consensus power less than $n + 1$ but *cannot* be solved using n-consensus objects? The answer is "no": our non-blocking emulation implies that every *terminating* protocol for $n + 1$ processes using objects A of consensus power less than $n + 1$ can be wait-free emulated using n-consensus objects. Thus, from the task-solvability perspective, n-consensus is the strongest object for systems of $n+1$ processes in which $(n + 1)$-process consensus is unachievable, i.e., in a strict sense, there are no objects between n- and $n + 1$-consensus.

The emulation technique presented in this paper is novel and interesting in its own right. It is based on the fundamental *inseparability* property which we show to be inherent for all $(n + 1)$-process protocols that use registers and objects of consensus power less than $n + 1$. Inseparability generalizes the concept of connectivity used in classical characterizations of distributed computing models (e.g., [6, 7, 4]) and it captures the very essence of the inability of a collection of shared objects to solve consensus.

Note that the aforementioned emulation does not answer the question of *robustness* of (deterministic) consensus hierarchy posed by Jayanti [9]. Our protocol does not imply that a protocol using a *composition* of deterministic objects A and B, each of consensus power n, can be emulated using only registers and n-consensus objects. Proving or refuting this statement is left for future work.

The paper is organized as follows. In Section 2, we describe the system model. In Section 3, we introduce the key notions of our result: inseparability and non-separating paths. Section 4 presents our emulation protocol and Section 5 concludes the paper.

2 Preliminaries

In this section, we describe the system model, and introduce some key notions. Missing details of the model can be found in [7, 9, 2].

Processes. We consider a set Π of $n + 1$ ($n \geq 1$) asynchronous processes $p_1, p_2, \ldots, p_{n+1}$ that communicate using atomic shared *objects*. Every object is characterized by a set of *ports* that it exports, a set of *states* the object can take, a set of *operations* that can be performed on the object, a set of *responses* that these operations can return, and a relation known as the *sequential specification* of the type that defines, for every state, port, and applied operation, the set of possible resulting states and returned responses. We assume that objects are *deterministic*: the sequential specification of a deterministic object is a function that carries each state, port, and operation to a new state and a corresponding response. In particular, we assume that processes have access to read-write shared memory.

Protocols. A *protocol* P is a collection of deterministic state machines $P_1, \ldots P_{n+1}$, one for each process. For every i, P_i maps every local state of p_i to the next operation it has to perform on a shared object (if any). Since all protocols and shared objects we consider are deterministic, we can model an *execution* of a protocol as an *initial* system state and a sequence of process identifiers, specifying the order in which processes take steps in the execution. To simplify the presentation, our emulation assumes that a protocol has only one initial state. In Section 4.3, we show how our emulation can be extended to protocols with multiple initial states.

We also distinguish between *terminating* and *non-terminating* protocols. In a terminating protocol, every process that takes sufficiently many steps reaches an irrevocable *final* view (we say that the process *terminates*). Without loss of generality, we consider protocols in which a terminated process keeps taking *null* steps, i.e., we focus on protocols in which every active process takes infinitely many steps.

States and views. Every finite execution is regarded as a *state of* P, unambiguously defining states of all shared objects and local states of processes (to be called *views*) that result after the execution completes.

There is a natural ancestor/descendant relation between states of P: we say that state x' is a *descendant* of state x (x is an *ancestor* of x'), and we write $x \rightarrow x'$, if x is a prefix of x'. We also say that a (finite or infinite) execution e *extends* a state x if x is a prefix of e.

We say that a view v of a process p_i *appears* in a state x (of P) if the local state of p_i in x is v. Let $\mathit{view}_i(x)$ denote the view of process p_i that appears in state x. We say that a view v is *compatible with a state* x if v appears in x or an ancestor of x.

For an execution e and a process p_i, $e|i$ denotes the sequence of distinct views of p_i (resulting after p_i takes its steps in e) that appear in prefixes of e (in the order of appearance). For two distinct views v and v' of the same process p_i, we say that v' is a descendant of v (or v precedes v'), and we write $v \rightarrow v'$, if there is an execution e such that v precedes v' in $e|i$.

For two views v and v' of the same process p_i such that $v \rightarrow v'$, $\mathit{next}(v, v')$ denotes the earliest view u such that (i) $v \rightarrow u$, and (ii) $u = v'$ or $u \rightarrow v'$. We say that v immediately precedes v' if $v \rightarrow v'$ and $v' = \mathit{next}(v, v')$.

Consensus. The *consensus* problem [6] is one in which a set of processes need to reach an agreement on one of their *proposed* values. More precisely, every process starts with an initial proposal in $\{0, 1\}$ and it is required that: (Termination) Every process that takes sufficiently many steps eventually decides on a value; (Agreement) No two processes decide on different values; (Validity) If a process decides on a value v, then v was proposed by some process.

It is sometimes convenient to relate the consensus problem among m processes to the *m-consensus object*. The object exports m ports and can be accessed with the *propose*() operation that takes a value as a parameter and returns one of the proposed values, so that no two *propose*() operations return different values.

We say that an object A *solves m-process consensus* if there exists an m-process protocol that solves consensus using read-write registers and copies of object A.[3] The

[3] The protocol designer is allowed to initialize the shared objects to any (reachable) states: this ability does not affect the consensus power of deterministic objects [3].

consensus power of A, denoted *cons*(A), is the largest m such that A solves m-process consensus [7]. If no such largest m exists, then *cons*$(A) = \infty$. Further, if *cons*$(A) = n$, then *cons*$(\{A, n\text{-}consensus\})$, the consensus power of the composition of A and n-process consensus, is n, i.e., $(n + 1)$-process consensus cannot be solved using copies of an object of consensus power n and n-consensus objects [4]

Team consensus is a form of consensus in which processes are divided a priori into two non-empty teams and which satisfies Validity, Termination, and *Team Agreement*: no two processes decide on different values, under the condition that processes on the same team propose the same value. Consensus is, in a precise sense, equivalent to *team consensus*: if A can solve m-process team consensus, then A can solve m-process consensus [11, 12].

Approximate agreement. Though it is impossible to reach non-trivial agreement using only read-write registers [6, 10], we can achieve *approximate* agreement [5] that guarantees that all decided values are sufficiently *close*. Formally, the ε-agreement task (where $\varepsilon \in [0, 1]$ is a specified parameter) is defined for two processes, q_0 and q_1, as follows. Every process q_i $(i = 0, 1)$ outputs a value $x_i \in [0, 1]$ such that (1) if q_i is the only participant, then q_i outputs i, and (2) $|x_0 - x_1| \leq \varepsilon$. It is known that, for all $\varepsilon \in (0, 1]$, the 2-process ε-agreement task is wait-free solvable using read-write registers [5].

Protocol emulation. We address the following problem: given a protocol P that uses objects in a set $\mathcal{C}$, design a protocol that *emulates* P using objects in a set $\mathcal{C}'$. In the emulation, processes start from their views in an initial state of P, and every active (not yet terminated) process may periodically output a new view that results after the process takes one more step of P.

On the safety side, the emulation must guarantee that all views output by the processes are compatible with *some* execution of P. On the liveness side, a *non-blocking* emulation ensures that either every participating process eventually reaches a final view, or at least one participant obtains infinitely many distinct views. Clearly, if P is terminating, then any non-blocking emulation of P is also *wait-free*: every participant eventually reaches a final view.

3 Inseparability

The following observation generalizes the arguments of most valence-based asynchronous impossibility proofs (e.g., [6, 7, 4]). Let P be any protocol using objects of collective consensus power n, and let x be any state of P. Then the immediate descendants of x are, in a strict sense, connected. More precisely, for every non-empty partitions Π_0 and Π_1 of Π, there exists an execution e of P going through x in which some process p_i takes infinitely many steps, without being able to decide whether the first step of e extending x was taken by a member of Π_0 or a member of Π_1.

Formally, let P be any protocol, and e_0 and e_1 be executions of P. We say that e_0 and e_1 are i-*confusing* if (1) p_i takes infinitely many steps in both e_0 and e_1, and (2) $e_0|i = e_1|i$. In other words, p_i cannot distinguish e_0 and e_1, even by taking infinitely many steps of P.

Let x_0 and x_1 be any two states of P. We say that x_0 and x_1 are *inseparable* (by P), and we write $x_0 \sim x_1$, if there exist $p_i \in \Pi$ and e_0 and e_1, extending x_0 and x_1,

respectively, such that e_0 and e_1 are i-confusing and either *view*$_i(x_0) =$ *view*$_i(x_1)$, or *view*$_i(x_0)$ immediately precedes *view*$_i(x_1)$, or vice versa.

Lemma 1. *Let x_0 and x_1 be any two states of a protocol P, such that $x_0 = x_1$ or $x_0 \rightarrow x_1$ and some process $p_i \in \Pi$ takes at most one step in x_1 after x_0. Then $x_0 \sim x_1$.*

Proof. Indeed, since p_i takes at most one step in x_1 after x_0, either *view*$_i(x_0) =$ *view*$_i(x_1)$ or *view*$_i(x_0)$ immediately precedes *view*$_i(x_1)$. Let e be any execution extending x_1 in which p_i takes infinitely many steps. Then $e_0 = e$ and $e_1 = e$ are i-confusing, and, thus, $x_0 \sim x_1$. □

3.1 Inseparably connected sets of states

Let $\approx$ denote the transitive closure of the $\sim$ relation. We say that a set of states is *inseparably connected* if, for every two states x_0 and x_1 in the set, $x_0 \approx x_1$.

Let x be any state of P. An *immediate descendant* of x is a one-step extension of x, i.e., a state that results after some process applies exactly one step to x. Note that if $x_0 = x_1$ or x_0 is an immediate descendant of x_1, then, by Lemma 1, $x_0 \sim x_1$, and if x_0 is a descendant of x_1, then $x_0 \approx x_1$. Let $G(x)$ denote the set of all $n+1$ immediate descendants of x. We say that a protocol P is *inseparably connected* if for every state x of P, $G(x)$ is inseparably connected.

Theorem 1. *Let A be any deterministic object of consensus power less than $n+1$ and P be any protocol among $n+1$ processes using copies of A and read-write registers. Then P is inseparably connected.*

Proof. Suppose, by contradiction, that there exists a state x of P, such that $G(x)$, the set of all $n+1$ immediate descendants of x, is not inseparably connected. We establish a contradiction by presenting a protocol that solves (team) consensus among $n+1$ processes using objects A and read-write registers.

Since $G(x)$ is not inseparably connected, it can be partitioned into two non-empty sets, G_0 and G_1, such that for all $x_0 \in G_0$ and $x_1 \in G_1$, $x_0 \not\approx x_1$. Let processes whose steps applied to x result in G_0 constitute team Π_0 and the rest constitute team Π_1. Clearly, $\Pi_0 \cup \Pi_1 = \Pi$.

Assume that all objects used by P are initialized to their states in x. Let R_0 and R_1 be two shared registers, initially $\perp$. Every process p_i first writes its proposal in a shared register R_j such that $p_i \in \Pi_j$. Then p_i takes steps of P starting from its view in x. Note that the views obtained by the processes are compatible with some execution of P extending x. The process stops when, for some $k \in \{0,1\}$, its view cannot appear in any descendant of any state in G_{1-k}. At this point, the process returns the value read in R_k.

Suppose, by contradiction, that the Termination property of consensus is violated: assume, without loss of generality, that P has an execution e_0 passing through a state $x_0 \in G_0$, in which p_i takes infinitely many steps, and every view obtained by p_i in e_0 could have been obtained in an execution e_1 passing through a state $x_1 \in G_1$. Thus, e_0 and e_1 are i-confusing. Moreover, since x_0 and x_1 are immediate descendants of the

same state x, *view*$_i(x_0) =$ *view*$_i(x_1)$, or *view*$_i(x_0)$ immediately precedes *view*$_i(x_1)$, or vice versa. Thus, $x_0 \sim x_1$ — a contradiction.

Now suppose that a process gets a view that is only compatible with descendants of states in G_k ($k \in \{0, 1\}$). Thus, the current execution e extends a state in G_k, and the process that took the first step in e after x has previously written its proposal in R_k. By the algorithm, the process returns the value read in R_k and, thus, Validity is satisfied. Since e extends a state in G_k, no process can ever obtain a view (in e) that is only compatible with executions extending a state in G_{1-k}. Thus, no process ever returns a value read in R_{1-k}. Now assume that processes on the same team (Π_0 or Π_1) propose the same value. Thus, no two different non-$\perp$ can be read in R_k — Team Agreement is ensured.

Hence, object A solves $(n + 1)$-team consensus and, therefore, $(n + 1)$-consensus — a contradiction. □

3.2 Non-separating paths

In every phase of our emulation protocol, processes try to reconcile their (possibly different) estimates of the emulated system state and the views to be output. Since we can use only registers and n-consensus objects, $n+1$ participants can only be guaranteed to reach *approximate* agreement. The approximate agreement is solved along a *path* connecting the concurrent estimates.

Formally, let P be a protocol, and (v_0, x_0) and (v_1, x_1) be tuples such that for $i = 0, 1$, x_i is a non-initial state of P, and v_i is a view that is compatible with x_i. Let *prec*$_i(x_0, x_1)$ ($i = 0, 1$) be defined as follows. If $x_0 \sim x_1$, then *prec*$_i(x_0, x_1) = x_i$. Otherwise, *prec*$_i(x_0, x_1)$ is the immediate predecessor of x_i.

We say that a sequence $(u_0, y_0), \ldots, (u_\ell, y_\ell)$, where each u_j is a view of P and each y_i is a state of P, is a *non-separating path connecting* (v_0, x_0) *and* (v_1, x_1) if:

(1) $(u_0, y_0) = (v_0, x_0)$ and $(u_\ell, y_\ell) = (v_1, x_1)$.
(2) $\forall j = 0, \ldots, \ell - 1$, u_j and u_{j+1} are both compatible with both y_j and y_{j+1}.
(3) $\forall j = 0, \ldots, \ell - 1$, $y_j \sim y_{j+1}$.
(4) $\forall j = 1, \ldots, \ell - 1$, $\exists i \in \{0, 1\}$, such that (i) *prec*$_i(x_0, x_1) \rightarrow y_j$ and (ii) either $u_j = v_i$ or u_j is not compatible with *prec*$_i(x_0, x_1)$.

Property (2) stipulates that the views of every two neighbors in a non-separating path must be compatible with both corresponding states. Intuitively, we need this property to ensure that all views produced by our emulation appear in some execution of the emulated protocol, i.e., the emulation is *safe*.

Property (3) requires that the states of every two neighbors in a non-separating path must be inseparable. Thus, it makes sure that competing state estimates produced by our emulation protocol are, in a strict sense, connected, so we could inductively extend the emulation. Property (4) implies that, for every view u_j on the path, there exists $i \in \{0, 1\}$ such that, unless $u_j = v_i$, u_j appears in a descendant of *prec*$_i(x_0, x_1)$ and does not appear in *prec*$_i(x_0, x_1)$. In other words, view u_j is "fresh" with respect to *prec*$_i(x_0, x_1)$. Intuitively, we need properties (3) and (4) to ensure that our emulation indeed makes progress, i.e., in each phase of the emulation, at least one participating process manages to perform one more step of P and obtain a new view.

If x'_0 and x'_1 are inseparable, then any two tuples (v_0, x_0) and (v_1, x_1) where each x_i $(i = 0, 1)$ is x'_i or one of x'_i's immediate descendants and v_i is compatible with x_i can be connected via a non-separating path. Moreover the *length* of this path (the number of hops) is bounded by $10(2n + 1)$.

Lemma 2. *Let P be any inseparably-connected protocol. Let x'_0 and x'_1 be non-initial states of P such that $x'_0 \sim x'_1$. Then for all (v_0, x_0) and (v_1, x_1) such that $\forall i = 0, 1$, $x_i \in \{x'_i\} \cup G(x'_i)$ and v_i is compatible with x_i, there exists a non-separating path connecting (v_0, x_0) and (v_1, x_1) the length of which does not exceed $\mathcal{L} = 10(2n + 1)$.*

Proof. There are three possible cases:
(a) Assume that $x_0 \sim x_1$ and, thus, $\mathit{prec}_j(x_0, x_1) = x_j$, $j = 0, 1$.

Let i be the smallest process identifier such that there exists i-confusing executions e_0 and e_1 that extend x_0 and x_1, respectively, and either $\mathit{view}_i(x_0) = \mathit{view}_i(x_1)$, or $\mathit{view}_i(x_0)$ immediately precedes $\mathit{view}_i(x_1)$, or $\mathit{view}_i(x_1)$ immediately precedes $\mathit{view}_i(x_0)$.

Let z_j $(j = 0, 1)$ be the shortest prefix of e_j in which p_i obtains a view that is compatible *neither* with x_0 *nor* with x_1. Note that, since e_0 and e_1 are i-confusing, $\mathit{view}_i(z_0) = \mathit{view}_i(z_1) = v$, and, since $\mathit{view}_i(x_0)$ and $\mathit{view}_i(x_1)$ are either identical or one of them immediately precedes the other, for each $j = 0, 1$, p_i takes at most two steps from x_j to reach view v.

Let y_j $(j = 0, 1)$ be the shortest prefix of e_j in which p_i takes at least one step after x_j (note that y_j can be equal to z_j). Let $u_j = \mathit{view}_i(y_j)$ (note that u_j can be equal to v).

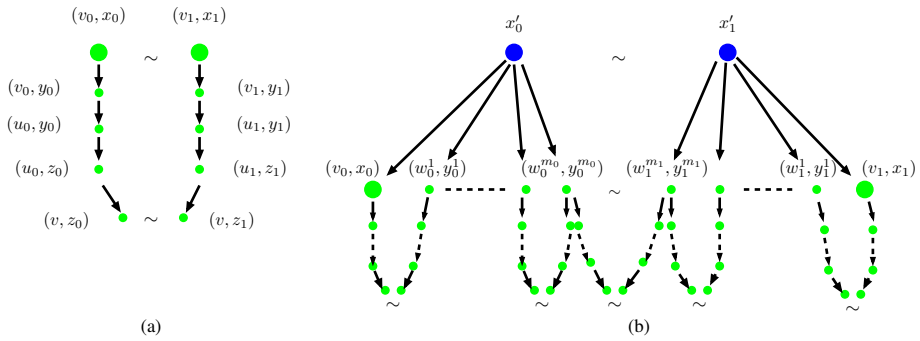

Fig. 1. Non-separating path connecting (v_0, x_0) and (v_1, x_1): (a) $x_0 \sim x_1$, (b) $x_0 \in G(x'_0)$ and $x_1 \in G(x'_1)$

Now we construct the non separating path connecting (v_0, x_0) and (v_1, x_1) as follows (Figure 1 (a)): (v_0, x_0), (v_0, y_0), (u_0, y_0), (u_0, z_0), (v, z_0), (v, z_1), (u_1, z_1), (u_1, y_1), (v_1, y_1), (v_1, x_1). Trivially, property (1) of non-separating paths are satisfied. Note that, by construction, e_0 and e_1 extend z_0 and z_1, respectively, and $\mathit{view}_i(z_0) = \mathit{view}_i(z_1) = v$. Since e_0 and e_1 are i-confusing, $z_0 \sim z_1$. By Lemma 1, $x_j \sim y_j$ and $y_j \sim z_j$ $(j = 0, 1)$. Thus, property (3) is satisfied.

For each internal vertex (u, y) in the path, u is compatible with the states of both its neighbors and, unless $u \in \{v_0, v_1\}$, u is not compatible with $x_j = \mathit{prec}_j(x_0, x_1)$

($j = 0, 1$). Finally, each internal state in the path (y_j or z_j) is a descendant of $x_j = \mathit{prec}_j(x_0, x_1)$. Thus, properties (2) and (4) are satisfied. The length of the path is 9.

(b) Now assume that $x_0 \nsim x_1$, $x_0 \in G(x'_0)$ and $x_1 \in G(x'_1)$, and, thus, $\mathit{prec}_i(x_0, x_1)$ is the immediate ancestor of x_i, $i = 0, 1$. Recall that $x'_0 \sim x'_1$. Let i be the smallest process identifier such that there exists i-confusing executions e_0 and e_1 that extend x'_0 and x'_1, respectively, and $\mathit{view}_i(x'_0)$ and $\mathit{view}_i(x'_1)$ are either identical or one of them precedes the other.

Let, for $j = 0, 1$, y'_j be the state resulting after the first step of e_j is applied to x'_j. Clearly, $y'_0 \in G(x'_0)$ and $y'_1 \in G(x'_1)$. Note that since executions e_0 and e_1 are i-confusing executions for x'_0 and x'_1, they are also i-confusing executions for y'_0 and y'_1. Let u'_j denote the last view of the process taking the last step in y'_j, $j = 0, 1$.

Similarly to case (a), let y_j ($j = 0, 1$) denote the shortest prefix of e_j in which p_i takes exactly one step after x'_j, and z_j denote the shortest prefix of e_j in which p_i obtains a view that is compatible neither with x'_0 nor with x'_1. Let u_j and be the last view of p_i in y_j. Note that since $x'_0 \sim x'_1$, $\mathit{view}_i(z_0) = \mathit{view}_i(z_1) = v$. Thus, $Z = (u'_0, y'_0)$, (u'_0, y_0), (u_0, y_0), (u_0, z_0), (v, z_0), (v, z_1), (u_1, z_1), (u_1, y_1), (u'_1, y_1), (u'_1, y'_1) is a non-separating path connecting (u'_0, y'_0) and (u'_1, y'_1).

Recall that P is inseparably connected, so $G(x'_0)$ and $G(x'_1)$ are each inseparably connected. Thus, for all $j = 0, 1$, there is a sequence of states $y_j^0, \ldots, y_j^{m_j}$ of $G(x'_j)$ such that $y_j^0 = x_j$, $y_j^{m_j} = y'_j$, and, for each $k = 0, \ldots, m_j - 1$, $y_j^k \sim y_j^{k+1}$. Let w_j^k denote the last view of the process taking the last step in y_j^k.

Now let, for each $j = 0, 1$ and $k = 0, \ldots, m_j - 1$, Z_j^k denote the non-separating path, constructed as described above in case (a), connecting (w_j^k, y_j^k) and (w_j^{k+1}, y_j^{k+1}).

Now consider the concatenation of paths Z_0^0, Z_0^1, …, $Z_0^{m_0-1}$, $Z_0^{m_0}$, Z, $Z_1^{m_1}$, $Z_1^{m_1-1}$, …, Z_1^1, Z_1^0 (Figure 1 (b)). Inductively, this is a non-separating path connecting (v_0, x_0) and (v_1, x_1). Since paths Z_j^k and Z are bounded by 9 and $m_j \leq n$ ($j = 0, 1$), the total length of the non-separating path is bounded by $9(2n+1) + 2n < 10(2n+1)$.

(c) Finally, assume that $x_0 \nsim x_1$, $x_0 = x'_0$ and $x_1 \in G(x'_1)$. Let x''_0 be the state in $G(x_0)$ such that there exists $x''_1 \in G(x'_1)$ and $x''_0 \sim x''_1$ (since P is inseparably connected, such x''_0 and x''_1 do exist). Let v''_j denote the last view of the process taking the last step in x''_j, $j = 0, 1$.

By employing the reasoning of case (b), we construct a non-separating path Z connecting (v''_0, x''_0) and (v_1, x_1). Since x''_0 and x''_1 are inseparable, the path does not "travel" through states in $G(x_0)$, and the length of Z is bounded by $9(n+1) + n$.

Then the non-separating path connecting (v_0, x_0) and (v_1, x_1) is obtained by adding (v_0, x_0), (v_0, x''_0) to the beginning of Z. The case $x_0 \in G(x'_0)$ and $x_1 = x'_1$ is symmetric. The length of the resulting path is bounded by $10(2n+1)$. □

3.3 P-reconciliation

To reconcile possibly conflicting estimates of the system state and the view to promote, our emulation of a protocol P uses a subroutine called *P-reconciliation*. In the 2-process P-reconciliation task, every process q_i ($i = 0, 1$) has an input (v_i, x_i), where v_i is a view of P and x_i is a state of P such that v_i is compatible with x_i. If both q_0 and q_1 participate, and there exists a non-separating path connecting (v_0, x_0) and (v_1, x_1)

of length at most $\mathcal{L}$ (let γ be the shortest such path), then each q_i outputs a tuple (U_i, z_i) where U_i is a set of at most two views and y_i is a state of P, such that:

(1) $U_0 \cap U_1 \neq \emptyset$, and
(2) there exist two neighbors (u_0, y_0) and (u_1, y_1) in γ such that $\{z_0, z_1\} \subseteq \{y_0, y_1\}$ and $U_0 \cup U_1 \subseteq \{u_0, u_1\}$.

Otherwise, if q_i is the only participant in the task or no such path γ exists, then q_i outputs $(\{v_i\}, x_i)$.

Shared variables :
  R_0, R_1, initially $\perp$

upon P-*reconcile*(v_i, x_i):

1: $R_i := (v_i, x_i)$
2: $s :=$ ε-*agreement*() $\{\varepsilon = 1/(2\mathcal{L})$ *where* $\mathcal{L} = 10(2n+1)\}$
3: **if** $R_{1-i} = \perp$ **then** {*If* q_i *goes solo* }
4: return $(\{v_i\}, x_i)$
5: $(v_{1-i}, x_{1-i}) := R_{1-i}$ {*Fetch the competing proposal*}
6: **if** (v_0, x_0) and (v_1, x_1) cannot be connected via a non-separating path of length $\leq \mathcal{L}$ **then**
7: return $(\{v_i\}, x_i)$
8: let $(u_0, y_0), (u_1, y_1) \ldots, (u_\ell, y_\ell)$ be the shortest non-separating path connecting (v_0, x_0) and (v_1, x_1)
9: let $j \in \{0, \ldots, \ell\}$ be such that $s \in ((j-1/2)/\ell, (j+1/2)/\ell]$
10: $z := y_j$
11: **if** $s \in ((j-1/4)/\ell, (j+1/4)/\ell]$ **then** {*If* s *belongs to* $1/4\ell$*-vicinity of* j/ℓ}
12: $U = \{u_j\}$
13: **else if** $s < j/\ell$ {s *belongs to the mid-half of* $[(j-1)/\ell, j/\ell]$}
14: $U = \{u_{j-1}, u_j\}$
15: **else** {s *belongs to the mid-half of* $[j/\ell, (j+1)/\ell]$}
16: $U = \{u_j, u_{j+1}\}$
17: return (U, z)

Fig. 2. 2-process P-reconciliation task: code for every process q_i, $i = 0, 1$

Lemma 3. *P-reconciliation is wait-free solvable using read-write registers.*

Proof. A P-reconciliation algorithm is presented in Figure 2. In the algorithm, every process q_i first registers its proposal (v_i, x_i) in the shared memory (line 1). Then it runs ε-agreement with $\varepsilon = 1/(2\mathcal{L})$ ($\mathcal{L} = 10(2n+1)$). Let $s \in [0, 1]$ be the value returned by ε-agreement. Then q_i reads R_{1-i} (line 3). If $R_{1-i} = \perp$ or there does not exist a non-separating path γ of length $\ell \leq \mathcal{L}$ connecting the two proposals, then q_i returns $(\{v_i\}, x_i)$ (lines 4 and 7). Otherwise, q_i computes a tuple (U_i, z_i), based on the output of ε-agreement (s), as follows (the procedure is summarized in Figure 3):

- Let $j \in \{0, \ldots, \ell\}$ be such that s belongs to a $1/(2\ell)$-neighborhood of j/ℓ, i.e., $s \in ((j-1/2)/\ell, (j+1/2)/\ell]$. Then p_i sets $z_i = y_j$.
- If s belongs to the $1/(4\ell)$-neighborhood of j/ℓ, i.e., $s \in ((j-1/4)/\ell, (j+1/4)/\ell]$ then p_i sets $U_i = \{u_j\}$.
- Otherwise, if $s < j/\ell$, i.e., s belongs to the middle half-interval of $[(j-1)/\ell, j/\ell]$, then p_i sets $U_i = \{u_{j-1}, u_j\}$. Else, if s belongs to the middle half-interval of $[j/\ell, (j+1)/\ell]$, then p_i sets $U_i = \{u_j, u_{j+1}\}$.

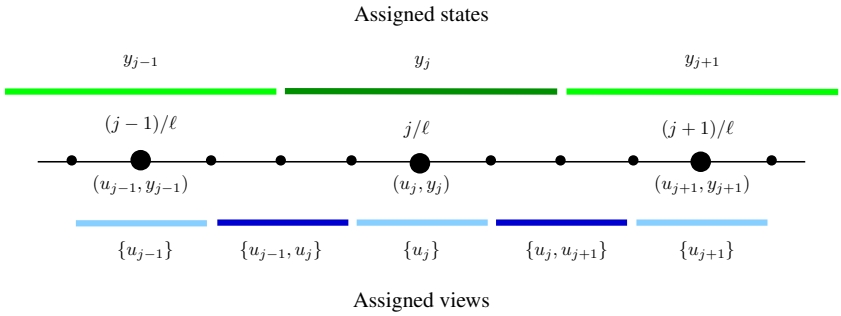

Fig. 3. Mapping the output of ε-agreement to views U_i and states y_i

Since $\varepsilon = 1/(2\mathcal{L})$ and $\ell \leq \mathcal{L}$, the values s_0 and s_1 returned by ε-agreement at q_0 and q_1, respectively, are within $1/(2\ell)$ from each other.

By inspecting Figure 3 and taking into accound that $|s_0 - s_1| \leq 1/(2\ell)$, we observe that the following cases are only possible.

If q_0 and q_1 return the same state y_j (s_0 and s_1 both belong to the $1/(2\ell)$-neighborhood of some j/ℓ), then either U_0 and U_1 are equal to the same non-empty set of views ($U_0 = U_1 \in \{\{u_{j-1}, u_j\}, \{u_j\}, \{u_j, u_{j+1}\}\}$), or U_0 and U_1 are related by containment ($\{U_0, U_1\} \subseteq \{\{u_{j-1}, u_j\}, \{u_j\}\}$ or $\{U_0, U_1\} \subseteq \{\{u_j, u_{j+1}\}, \{u_j\}\}$). In both cases, the set of all returned views is either a subset of $\{u_{j-1}, u_j\}$ or or a subset of $\{u_j, u_{j+1}\}$, and the properties of P-reconciliation are satisfied.

If q_0 and q_1 return different states, then the two states can only be some y_j and y_{j+1} (neighbors in γ). Furthermore, either q_0 and q_1 both return $\{u_j, u_{j+1}\}$ as the sets of views, or one of them returns $\{u_j, u_{j+1}\}$, and the other — $\{u_j\}$ or $\{u_{j+1}\}$, and the properties of P-reconciliation are satisfied. □

4 Protocol emulation

Let P be any inseparably connected $(n+1)$-process protocol. In Figure 4, we present a non-blocking emulation of P that uses only n-consensus objects and read-write registers.

4.1 Overview

In the emulation, every process periodically outputs views, called *converged views*. The emulation guarantees that at every moment of time, there exists an execution of P that is compatible with the sequence of views output locally at every process, and either every participating process eventually reaches a final view, or at least one participant outputs infinitely many views.

The emulation proceeds in asynchronous *phases*. The participants of a phase first split into two teams, q_0 and q_1. If $n = 1$, then p_i is assigned to team q_i, $i = 0, 1$. If

$n \geq 2$, then every process dynamically chooses its team using a sequence of n test-and-set objects.[4] Every process that wins one of the test-and-set objects joins the first team, denoted q_0, and the left-out process constitutes the second team, denoted q_1. Note that if $n \geq 2$ and n or less processes participate in a phase, then there can be at most active team in that phase — q_0.

Each team q_j then agrees on the estimate of the emulated system state and a "fresh" view of a member of q_j that q_j is willing to promote in the current phase (in case $n \geq 2$, team q_0 uses an n-consensus object for this).

```
Shared variables :
    n-consensus objects C[ ][ ]    {An array of consensus objects for every phase}
Important local variables at p_i:
    x_i, initially x̄    {Current system state estimate (x̄ is the initial system state)}
    v_i, initially view_i(x̄)    {The last converged view of p_i}
1:  k := 0
2:  repeat
3:    k := k + 1
4:    if v_i = view_i(x_i) then
5:      z_i := the state after p_i applies its step to x_i
6:    else
7:      z_i := x_i
8:    w_i := next(v_i, view_i(z_i))    {Choose the next view of p_i to promote}
9:    join a team q_j (j = 0, 1)    {If n ≥ 2, first n join q_0 using test-and-set objects,}
                                    {and the last joins q_1; if n = 1, p_i joins team q_i  i = 0, 1}
10:   (w, z) := agree with q_j on (w_i, z_i)    {If n ≥ 2, then q_0 uses n-consensus}
11:   (U, y) := P-reconcile_k(w, z)    {On behalf of team q_j, using n-consensus}
12:   x_i := y
13:   if ∃u ∈ U, v_i → u then    {A view of p_i has converged in phase k}
14:     v_i := next(v_i, u)    {Compute the next view of p_i toward u}
15:     output v_i
16: until v_i is final
```

Fig. 4. The emulation protocol: code for every process p_i, $i = 1, \ldots, n+1$

The two teams then act as processes q_0 and q_1 in an instance of P-reconciliation task which they use to reconcile on how to make progress. If $n \geq 2$, then in every instance of P-reconciliation, multiple processes in team q_0 use a series of n-consensus objects (distinct for every instance of P-reconciliation) to act as a single process. Team q_0 is associated with an $(n+1)$-array of registers T_0, and q_1 is associated with a register T_1. To write a value v in executing the ε-agreement protocol, every process in q_0 increments its local sequence number and updates its slot in T_n with v, equipped with the monotonically increasing sequence number, while q_1 simply writes the value in T_1. To perform a read operation, processes in q_0 read T_1 and use n-process consensus to agree on the read value, while q_1 reads all slots in T_0 and returns the value with the largest sequence number.

Let the P-*reconcile*$_k(w, z)$ procedure return a tuple (U, y) at process p_i (line 11). Then p_i adopts y as its state estimate, and if U contains a view u of p_i, then p_i outputs the next view toward u (line 15) . We demonstrate below that, inductively, all views

[4] Test-and-set objects can be wait-free implemented from 2-process consensus objects and thus from objects of consensus power 2 [1].

output up to phase k are compatible with every state estimate computed at the end of phase k. Furthermore, by the properties of P-reconciliation, all views in U are "fresh", i.e., they extend views output in previous phases. Since the sets of views returned by P-reconciliation have a non-empty intersection, at least one process p_i participating in every phase is guaranteed to output a new view at the end of the phase. Thus, the resulting simulation is both safe and live.

4.2 Correctness

Theorem 2. *Every inseparably connected $(n+1)$-process protocol P can be emulated in a nonblocking manner using n-consensus objects and read-write registers.*

Proof. Consider the protocol in Figure 4. We show first that the protocol preserves the following invariants:

- (**P1**) In each phase k, there exist two inseparable states $\tilde{x}_0$ and $\tilde{x}_1$ of P such that each system state estimate x_i computed at the end of phase k is in $\{\tilde{x}_0, \tilde{x}_1\}$, and all views output up to phase k are compatible with both $\tilde{x}_0$ and $\tilde{x}_1$.
- (**P2**) At the end of each phase k, at least one process p_i *that participated in the phase* (i.e., reached line 9 in Figure 4 in phase k) succeeds in taking one more emulated step of P and outputs a new view.

Initially, all processes agree on the initial state of P, denoted $\bar{x}$, and the initial views of all processes are compatible with $\bar{x}$. Inductively, suppose that P1 holds at the end of phase $k-1$. In phase k, every process p_i chooses the next state z_i and the next view w_i (that p_i obtains if it is chosen to make a new step of P), based on its current view v_i and its current system state estimate x_i (lines 5 and 8). Note that each $z_i \in \{\tilde{x}_0, \tilde{x}_1\} \cup G(\tilde{x}_0) \cup G(\tilde{x}_1)$, and w_i is compatible with z_i. Since $\tilde{x}_0 \sim \tilde{x}_1$, $\mathit{prec}_j(z_0, z_1) \in \{\tilde{x}_0, \tilde{x}_1\}$ $(j = 0, 1)$.

Let $(\tilde{w}_j, \tilde{z}_j)$ denote the tuple proposed by team q_j $(j = 0, 1)$ to the P-reconciliation procedure in line 10, and let (U_j, y_j) be the value returned by $P\text{-}\mathit{reconcile}_k(\tilde{w}_j, \tilde{z}_j)$. By Lemma 2, there exists a non-separating path connecting $(\tilde{w}_0, \tilde{z}_0)$ and $(\tilde{w}_1, \tilde{z}_1)$ of length $\leq \mathcal{L}$. By the properties of P-reconciliation, (U_0, y_0) and (U_1, y_1) correspond to some neighbors in such a path (let us denote it γ).

Thus, by the properties of non-separating paths, y_0 and y_1 are inseparable, and every view $U_0 \cup U_1$ is compatible with both y_0 and y_1. Further, each y_j extends $\tilde{x}_0$ or $\tilde{x}_1$, and each $u \in U_0 \cup U_1$ is *not* compatible with some $\tilde{x}_0$ or $\tilde{x}_1$, unless $u \in \{\tilde{w}_0, \tilde{w}_1\}$. Thus, every view that is compatible with both $\tilde{x}_0$ and $\tilde{x}_1$ is also compatible with both y_0 and y_1. Since each $p_i \in q_j$ sets x_i to y_j (line 12) and outputs a new view only if it belongs to U_j (line 14), P1 is inductively preserved.

Now, since $U_0 \cap U_1 \neq \emptyset$, there is at least one view that is seen by every process that completes the phase. Thus, at least one process p_i will find its view in U_j. Thus, at least one process will output a new view in line 15. Note that if n or less process participate in phase k (which can happen only if some process crashes in phase $k-1$ or earlier), then only one team (if $n \geq 2$, then it can only be q_0) takes part in P-reconciliation and, thus, the participants of phase k agree on the state and the view proposed by one of them. Otherwise, if all $n+1$ processes participate in phase k, *some* process in Π gets

a new view. In both cases, some process *participating* in phase k obtains a new view. Thus, P2 is preserved.

Finally, P1 and P2 ensure that in every execution of our emulation, all output views are compatible with some execution of P, and, at least one active process keeps making progress by outputting new views. Thus, our protocol indeed emulates P in a non-blocking manner. □

Theorems 1 and 2 imply the following results:

Corollary 1. *Let A be a deterministic object of consensus power less than $n+1$. Every protocol using copies of A and registers can be emulated in a non-blocking manner using n-consensus objects and registers.*

Corollary 2. *Let A be a deterministic object of consensus power less than $n+1$. There is no task T that can be wait-free solved using copies of A and registers, but cannot be wait-free solved using n-consensus objects and registers.*

4.3 Multiple initial states

The emulation protocol presented in the previous section can be easily extended to protocols with multiple initial states that differ only in inputs of processes. (This seems to be the case for most protocols.) Indeed, in our emulation, a team q_i needs to know the initial states of other processes only if, in a given phase k, it faces a competition with the other team q_{1-i}, i.e., only if q_i reaches line 5 of the P-reconcile algorithm in Figure 2. If this never happens, then q_i only needs to know the (estimated) states of shared objects to be able to make progress. But such a competition between q_0 and q_1 can only happen if all $n+1$ processes participate in phase k (otherwise, only one team would be "populated"). Thus, we can easily extend our emulation to the case of multiple initial states: every process registers its input in the shared memory before participating in the first phase of the emulation protocol. When q_i faces the competition with the other team (q_i reaches line 5 in Figure 2), q_i can compute an estimate of the current state of P using all $n+1$ registered inputs. From this point on, the emulation will run as described in Figure 4.

5 Conclusions

We formalized one outcome of the intuition that n processes using n-process consensus are tantamount to a single process. Hence $n+1$ processes with n-process consensus are like two processes. It is easy to see why there is no task between read-write shared memory and 2-process consensus: A task's *output complex* [8] is either connected - then the two process can solve the task using reads and writes - or it is disconnected - in which case the task amounts to consensus. In generalizing this intuition we encountered certain difficulties: we only managed to equate n processes with one when a protocol was given ahead of time, and we could derive only a non-blocking emulation of the protocol. We conjecture that these limitations are inherent. We also conjecture that our results can be extended from $n+1$ to any number of processes between $n+1$ and $2n$: any deterministic object of consensus power less than $n+1$, when used to solve a task T for k processes, $n+1 \leq k \leq 2n$, can be replaced with n-consensus objects.

Acknowledgments

Special thanks should go to Rachid Guerraoui for the observation that our emulation establishes the distinction between n- and $(n + 1)$-consensus in determinstic shared memory models.

References

1. Y. Afek, E. Weisberger, and H. Weisman. A completeness theorem for a class of synchronization objects (extended abstract). In *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 159–170, 1993.
2. H. Attiya and J. L. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition)*. Wiley, 2004.
3. E. Borowsky, E. Gafni, and Y. Afek. Consensus power makes (some) sense! In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 363–372, August 1994.
4. T. D. Chandra, V. Hadzilacos, P. Jayanti, and S. Toueg. Generalized irreducibility of consensus and the equivalence of t-resilient and wait-free implementations of consensus. *SIAM J. Comput.*, 34(2):333–357, 2004.
5. D. Dolev, N. A. Lynch, S. S. Pinter, E. W. Stark, and W. E. Weihl. Reaching approximate agreement in the presence of faults. *J. ACM*, 33(3):499–516, 1986.
6. M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(3):374–382, April 1985.
7. M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.
8. M. Herlihy and N. Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858–923, November 1999.
9. P. Jayanti. On the robustness of Herlihy's hierarchy. In *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 145–158, August 1993.
10. M. C. Loui and H. H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, pages 163–183, 1987.
11. G. Neiger. Failure detectors and the wait-free hierarchy. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 100–109, August 1995.
12. E. Ruppert. *The Consensus Power of Shared-Memory Distributed Systems*. PhD thesis, University of Toronto, 1999.