# A separation of (n -1)-consensus and n-consensus in read-write shared-memory systems

Carole Delporte-Gallet, Hugues Fauconnier, Sam Toueg

# A separation of $(n-1)$-consensus and $n$-consensus in read-write shared-memory systems

Carole Delporte-Gallet
Université Paris Diderot
Paris, France

Hugues Fauconnier
Université Paris Diderot
Paris, France

Sam Toueg
University of Toronto
Toronto, ON, Canada

**Abstract**

A fundamental research theme in distributed computing is the comparison of systems in terms of their ability to solve basic problems such as consensus that cannot be solved in completely asynchronous systems. In particular, in a seminal work [**?**], Herlihy compares shared-memory systems in terms of the shared objects that they have: he proved that there are shared objects that are powerful enough to solve consensus among $n$ processes, but are too weak to solve consensus among $n+1$ processes; such objects are placed at level $n$ of a *wait-free hierarchy*. The importance of this hierarchy comes from Herlihy's universality result: intuitively, every object at level $n$ of this hierarchy can be used to implement *any* object shared by $n$ processes in a wait-free manner.

As in [**?**], we compare shared-memory systems with respect to their ability to solve consensus among $n$ processes. But instead of comparing systems defined by the shared objects that they have, we compare read-write systems defined by the *process schedules* that they allow. These systems capture a large set of systems, e.g., systems whose synchrony ranges from fully synchronous to completely asynchronous, several systems with failure detectors, and "obstruction-free" systems. In this paper, we consider read-write systems defined in terms of process schedules, and investigate the following natural question: For every $n$, is there a system of $n$ processes that is strong enough to solve consensus among every subset of $n-1$ processes in the system, but not enough to solve consensus among all the $n$ processes of the system? We show that the answer to the above question is "yes", and so these systems can be classified into hierarchy akin to Herlihy's hierarchy.

# 1 Motivation and related work

A fundamental research theme in distributed computing is the comparison of systems in terms of their ability to solve basic problems such as consensus or $k$-set agreement that cannot be solved in completely asynchronous systems [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**]. In particular, in a seminal work [**?**], Herlihy compares shared-memory systems in terms of the shared objects that they have: he proved that there are shared objects that are powerful enough to solve consensus among $n$ processes, but are too weak to solve consensus among $n+1$ processes; such objects are placed at level $n$ of a *wait-free hierarchy*. The importance of this hierarchy comes from Herlihy's universality result: intuitively, every object at level $n$ of this hierarchy can be used to implement *any* object shared by $n$ processes in a wait-free manner.

As in [**?**], in this paper we compare shared-memory systems with respect to their ability to solve consensus among $n$ processes. But instead of comparing systems defined by the shared objects that they have, we compare systems (with shared registers) defined by the *process schedules* that they allow, as we explain below.

First, note that several types of read-write shared-memory systems, e.g, *asynchronous*, *partially synchronous* and *synchronous* systems, can be defined by the set of process schedules that they allow.[1] For example, a completely asynchronous system is one where every process schedule can occur. Similarly, a partially synchronous system is one where the process schedules satisfy some timeliness or "fairness" conditions [**?**, **?**, **?**] which effectively define the set of process schedules that are allowed. Perfectly synchronous systems can also be defined by the set of process schedules that are allowed. Furthermore, several systems with *failure detectors* [**?**] are equivalent to systems defined in terms of process schedules: for several well-known failure detectors $\mathcal{D}$ (including $P, \diamond P, S$ and $\diamond S$) an asynchronous system augmented with $\mathcal{D}$ is *equivalent* to a system where schedules satisfy some fairness conditions [**?**].[2] Finally, *obstruction-free* algorithms work in systems with a specific set of process schedules, namely, schedules where some process eventually executes solo [**?**].

Thus, shared-memory systems defined in terms of process schedules capture a large set of systems, e.g., systems whose synchrony ranges from fully synchronous to completely asynchronous, several systems with failure detectors, and "obstruction-free" systems. In this paper, we consider such systems and investigate the following natural question:

> For every $n$, is there a system of $n$ processes that is strong enough to solve consensus among every subset of $n-1$ processes in the system, but not enough to solve consensus among all the $n$ processes of the system?

If this is true, it would imply that these systems can be classified into hierarchy akin to Herlihy's hierarchy.

The answer to the above question is not obvious. In [**?**] it is shown that if a failure detector $\mathcal{D}$ is powerful enough to solve consensus among every subset of $n-1$ processes in a system of $n$ processes, then it is powerful enough to solve consensus among all the $n$ processes in the system. Since [**?**] shows that several systems with failure detectors are equivalent to systems with sets of schedules, it is tempting to conjecture that the answer is "no":

In this paper we show that the answer to the above question is "yes". More precisely, we prove that for every $n \geq 2$, there is a read-write shared-memory system $S$ of $n$ processes such that: (a) consensus can be solved for every subset of $n-1$ processes of $S$, and (b) consensus cannot be solved for the $n$ processes of $S$. It is worth noting that the positive result (a) holds even if $S$ has only a bounded number of bounded-size registers, while the impossibility result (b) holds even if $S$ has an unbounded number of unbounded-size registers.

[1] Intuitively, a process schedule specifies the order in which processes take steps.

[2] The results in [**?**] were for message-passing systems, but similar results hold for read-write shared-memory systems.

# 2 Model

We consider shared-memory systems of processes with SWMR multivalued atomic registers. Processes proceed by executing atomic events: in each event, a single process can read or write a single register. In the following, $P$ denotes a set of processes, formally $P \subseteq \mathbb{N} = \{1, 2, \ldots\}$.

## 2.1 Process schedules

A *schedule* $\sigma$ of a set of processes $P$ is a finite or infinite sequence where each element of the sequence is a process in $P$, e.g., $\sigma = 243125434253335\ldots$ is a schedule of $P = \{1, 2, 3, 4, 5\}$. Each occurrence of a process $p$ in a schedule $\sigma$ is called a *step of $p$ (in $\sigma$)*. A process is *correct in a schedule* $\sigma$ if it occurs infinitely often in $\sigma$, otherwise it is *faulty (or crashes) in* $\sigma$.

Roughly speaking, a schedule $\sigma$ is $k$-*solo*, if $\sigma$ has a process that runs solo for at least $k$ consecutive steps infinitely often. More precisely, *a schedule $\sigma$ of a set of processes $P$ is $k$-solo* if $\sigma$ is finite or there is a process $p \in P$ such that $\sigma$ contains an infinite number of subsequences consisting of $k$ or more consecutive steps of $p$ (and only of $p$). Note that if a schedule is $k$-*solo* then it is also $(k-1)$-*solo*, and every schedule is trivially 1-*solo*.

## 2.2 Systems and subsystems

Intuitively, a system $S$ is described by the set $P$ of processes in the system and the set $\Sigma$ of schedules of $P$ that can occur in this system. More precisely, a system $S$ is a tuple $[P, \Sigma]$, where $P$ is a non-empty set of processes and $\Sigma$ is a non-empty set of schedules of $P$. We say that *$\sigma$ is a schedule of system $S = [P, \Sigma]$* if $\sigma \in \Sigma$. Moreover, *$S' = [P', \Sigma']$ is a subsystem of $S = [P, \Sigma]$* (denoted $S' \subseteq S$) if $P' \subseteq P$ and $\Sigma' \subseteq \Sigma$. Finally, $S' = [P', \Sigma']$ is a *proper* subsystem of $S = [P, \Sigma]$ (denoted $S' \subset S$) if $S'$ is a subsystem of $S$ and $P' \subset P$ or $\Sigma' \subset \Sigma$.

In the following definitions, $P$ is a set of processes and $Q$ is a subset of $P$. If $\sigma$ is a schedule of $P$, we denote by $\sigma(Q)$ the subsequence of $\sigma$ obtained by keeping only the steps of the processes that are in $Q$; e.g., if $\sigma = 243125434253335$ and $Q = \{2, 5\}$ then $\sigma(Q) = 225255$. Note that $\sigma(Q)$ is a schedule of $Q$. If $\Sigma$ is a set of schedules of $P$, we denote by $\Sigma(Q)$ the set of schedules obtained by keeping only the steps of the processes that are in $Q$ in the schedules of $\Sigma$; more precisely: $\Sigma(Q) = \{\sigma' \mid \exists \sigma \in \Sigma \text{ such that } \sigma' = \sigma(Q)\}$. Note that all the schedules of $\Sigma(Q)$ are schedules of $Q$.

Let $S = [P, \Sigma]$ be a system and $Q$ be a subset of $P$. We denote by $S(Q)$ the subsystem of $S$ obtained by keeping only the steps of the processes that are in $Q$ from $S$; more precisely, $S(Q) = [Q, \Sigma(Q)]$. Note that if $Q$ is a proper subset of $P$, then $S(Q)$ is a proper subsystem of $S$.

## 2.3 Systems $A_n$ and $S_n^k$

We now define some systems that are central to our results. Henceforth, $P_n$ is the set of $n$ processes $\{1, 2, \ldots, n\}$ and $\Sigma_n$ is the set of *all* the schedules of $P_n$.

- $A_n = [P_n, \Sigma_n]$ is *the asynchronous system of the processes in $P_n$*.
- For $k \geq 1$, $S_n^k = [P_n, \Sigma_n^k]$ where $\Sigma_n^k = \{\sigma \mid \sigma \in \Sigma_n$ and for all the proper subsets $Q$ of $P_n : \sigma(Q)$ is $k$-$solo\}$.

**Observation 1** *Let $Q$ be any proper subset of $P_n$. All the schedules of the subsystem $S_n^k(Q)$ of $S_n^k$ are $k$-solo schedules of $Q$.*

To see this, consider any schedule $\sigma'$ of $S_n^k(Q) = [Q, \Sigma_n^k(Q)]$, i.e., $\sigma' \in \Sigma_n^k(Q)$. So $\sigma' = \sigma(Q)$ for some $\sigma \in \Sigma_n^k$. By the definition of $\Sigma_n^k$, $\sigma(Q)$ (and therefore $\sigma'$) is a $k$-*solo* schedule of $Q$.

Note that:

1. For all $k \geq 1$, $S_n^1 = A_n$, the asynchronous system of the processes in $P_n = \{1, 2, \ldots, n\}$. This is because for every schedule $\sigma$ of $P_n$ and every proper subset $Q$ of $P_n$, the schedule $\sigma(Q)$ is trivially 1-$solo$.

2. For all $k \geq 1$, $S_2^k = A_2$, the asynchronous system of the processes in $P_2 = \{1, 2\}$. This is because for every schedule $\sigma$ of $P_2 = \{1, 2\}$ and every proper subset $Q$ of $\{1, 2\}$ (namely, $Q = \{1\}$, $Q = \{2\}$ or $Q = \emptyset$) the schedule $\sigma(Q)$ is trivially $k$-$solo$ for all $k \geq 1$.

3. For all $n \geq 3$, and all $k \geq 2$, $S_n^k \subset A_n$. To see this, let $k \geq 2$ and consider the infinite schedule $\sigma = 123123123123 \ldots etc.$ of $P_n = \{1, 2, \ldots, n\}$. This is a schedule of $A_n$, but it is not a schedule of $S_n^k$. This because for the proper subset $Q = \{1, 2\}$ of $P_n = \{1, 2, \ldots, n\}$, the schedule $\sigma(Q) = 12121212 \ldots etc.$ is not $k$-$solo$ for $k \geq 2$, so $\sigma$ is not a schedule of $S_n^k$.

4. For all $n \geq 3$, $A_2 \not\subseteq S_n^2$. To see this, consider the infinite schedule $\sigma = 1212121212 \ldots etc.$ This is a schedule of $A_2$, but it is not a schedule of $S_n^2$. This is because, for all $n \geq 3$, $Q = \{1, 2\}$ is a proper subset of $P_n = \{1, 2, \ldots., n\}$ but $\sigma(Q) = 1212121212 \ldots etc.$ is not 2-$solo$.

5. For all $n \geq 3$, and all $k \geq 1$, $S_n^{k+1} \subset S_n^k$. To see this, let $\sigma \in S_n^{k+1}$. For every $Q \subset P_n$, $\sigma(Q)$ is $(k+1)$-$solo$, and so $\sigma(Q)$ is also $k$-$solo$; thus $\sigma \in S_n^k$. Furthermore, consider the schedule $\sigma = \prod_{i=1}^{\infty}(1^k 2^k \ldots n^k)$. It is clear that for every $Q \subset P_n$, $\sigma(Q)$ is $k$-$solo$, and so $\sigma \in S_n^k$; but for $Q = \{1, 2\}$, $\sigma(Q) = 1^k 2^k 1^k 2^k \ldots 1^k 2^k \ldots$ is not $(k+1)$-$solo$, and so $\sigma \notin S_n^{k+1}$.

To provide some intuition about the systems $S_n^k$ that we defined, we now give a few simple examples of schedules that are in $S_n^k$ and are not in $S_n^k$.[3] To describe these schedules we use the following notation. For two processes $p$ and $q$ in $P_n = \{1, 2, \ldots, n\}$, $(pq)^i$ denotes the sequence of steps $pq$ repeated $i$ times; for example $(pq)^3$ is $pqpqpq$. Similarly, $\{p, q\}^i$ denotes *any* finite sequence of steps of processes $p$ and $q$ that contains *at least* $i$ steps of $p$ and *at least* $i$ steps of $q$, in any order. For example, $\{p, q\}^3$ includes the sequences $pqpqpq$, $pppqqqq$ and $ppqqpq$, but it does *not* include the sequence $ppprqqq$ (because it contains a step of process $r$) or $ppppqq$ (because it contains fewer than 3 steps of $q$).

We start with some examples of schedules of system $S_n^k$ (i.e., of schedules $\sigma \in \Sigma_n^k$) for $n \geq 3$:

(a) $\sigma = \prod_{i=1}^{\infty}[(12)^k(13)^k(14)^k \ldots (1n)^k]$. Schedule $\sigma \in \Sigma_n^k$ because it is a schedule of $P_n$ and for every proper subset $Q$ of $P_n$, the schedule $\sigma(Q)$ is $k$-$solo$. To see this, assume $Q$ is not empty (if $Q = \emptyset$, $\sigma(Q)$ is trivially $k$-$solo$ because it is the empty schedule). There are two possible cases. If $Q$ does not contains process 1, then, since $Q$ is not empty, $Q$ contains at least one process $j \in \{2, 3, \ldots n\}$; since the sequence $(1j)^k$ appears infinitely often in $\sigma$ and $1 \notin Q$, the sequence $j^k$ (i.e., $k$ consecutive steps of process $j$) appears infinitely often in schedule $\sigma(Q)$; thus $\sigma(Q)$ is $k$-$solo$. If $Q$ contains process 1, then, since $Q$ is a proper subset of $P_n$, some process $j \in \{2, 3, \ldots n\}$ is *not* in $Q$; since the sequence $(1j)^k$ appears infinitely often in $\sigma$ and $j \notin Q$, the sequence $1^k$ appears infinitely often in schedule $\sigma(Q)$; thus $\sigma(Q)$ is $k$-$solo$. Thus, in both cases $\sigma(Q)$ is $k$-$solo$.

(b) $\sigma$ is any schedule of the form $\prod_{i=1}^{\infty}[\{1, 2\}^k\{1, 3\}^k\{1, 4\}^k \ldots \{1, n\}^k]$. The proof that $\sigma \in \Sigma_n^k$ is similar to the one given above.

(c) $\sigma = \prod_{i=1}^{\infty}[(12)^k(23)^k(34)^k \ldots (n-2\ n-1)^k(n-1\ n)^k(n\ 1)^k]$. It is easy to see that for every $Q \subset P_n$, the schedule $\sigma(Q)$ is $k$-$solo$, so $\sigma \in \Sigma_n^k$.

[3] The examples of schedules that we give here are very simple (they have a simple repetitive pattern). It should be clear, however, that the set of schedules of $S_n^k$ is very "rich": it contains schedules that are much more varied and complex than the few simplistic ones that we give for illustration here.

The following schedule is not in system $S_n^k$, i.e., $\sigma \notin \Sigma_n^k$, for any $k \geq 2$ and $n$ even, $n \geq 4$ :

(a) $\sigma = \prod_{i=1}^{\infty}[(12)^k(34)^k \ldots (n-3\ n-2)^k(n-1\ n)^k]$. To see that $\sigma \notin \Sigma_n^k$, note that for the proper subset $Q = \{1, 2\}$ of $P_n$, the schedule $\sigma(Q) = 121212121212\ldots$ is not $k$-*solo* for any $k \geq 2$.

### 2.4 Consensus problem

Consider an algorithm $\mathcal{A}$ defined for some set of processes $P$. A schedule determines exactly the sequence of steps of the processes executing the algorithm. Then the set of runs of algorithm $\mathcal{A}$ for a system $S = [P, \Sigma]$ is the set of all runs of $\mathcal{A}$ obtained when the processes of $P$ execute $\mathcal{A}$ for $\Sigma$.

In the consensus problem each process has an initial value and must decide a value. We say that algorithm $\mathcal{A}$ solves the consensus in system $S = [P, \Sigma]$ if and only if every run of $\mathcal{A}$ for $S$ satisfies the following properties:

- *(Agreement)* If a correct process $p$ decides $v$ and a correct process $q$ decides $v'$ then $v = v'$;
- *(Integrity)* If some correct process decides $v$ then $v$ is the initial value of some process.
- *(Termination)* Every correct process eventually decides some value.

By a slight extension we say that $\mathcal{A}$ solves the consensus for a subset $Q$ of $P$ in $S = [P, \Sigma]$ if $\mathcal{A}$ solves the consensus in the subsystem $S(Q)$.

We sometimes consider the uniform variant of the consensus. In the uniform consensus, each run has to ensure (i) (*uniform agreement*) if a process $p$ decides $v$ and a process $q$ decides $v'$ then $v = v'$, (ii) (*uniform integrity*) if some process decides $v$ then $v$ is the initial value of some process, and (iii) *termination* as before.

## 3 Main result statement and proof outline

Our main result is that for all $n \geq 2$ there is a shared-memory system $S$ of $n$ processes that separates the problems of (1) solving consensus for $n$ processes and (2) solving consensus for every proper subset of these processes. More precisely, we show the following:

**Theorem 2** *For every $n \geq 2$, there is a system $S$ of processes $P_n$ such that:*

*(a) for every proper subset $Q$ of $P_n$, consensus can be solved in the subsystem $S(Q)$ of $S$, and*

*(b) consensus cannot be solved in $S$.*

**Proof outline.** For $n = 2$, the proof is obvious. $S$ is simply the asynchronous system of $P_2 = \{1, 2\}$, i.e., $S = A_2$. By the well-known impossibility of result of [**?**, **?**], consensus cannot be solved in $S$. Furthermore, for every proper subset $Q$ of $P_2 = \{1, 2\}$, consensus can be trivially solved in the subsystem $S(Q)$ of $S$ since this system contains at most one process.

For $n \geq 3$, the system $S$ is defined as follows. Consider the algorithm $\mathcal{B}_n$ for processes $P_n$ given in Section **??**, this algorithm using snapshots may be rewritten into an algorithm $\mathcal{B}'_n$ using only SWMR registers. Let $\ell$ be the number of steps that any process $p \in P_n$ takes to execute the main loop of this algorithm (i.e., Figure **??**, lines **??**-**??**) $2n + 1$ times. From the code of this loop, it is clear that $\ell$ is well-defined: this is because (a) the code does not contain any "wait" statement, and (b) multi-writer snapshot used in the algorithm can be wait-free implemented with a bounded number of SWMR registers such that each scan or update of a process $p$ terminates if $p$ takes more than $k$ steps alone for some $k$.

**Definition 3** *Let $S = S_n^{\ell}$.*

Recall that $S_n^\ell = [P_n, \Sigma_n^\ell]$ where $\Sigma_n^\ell = \{\sigma \mid \sigma \in \Sigma_n$ and for all $Q \subset P_n : \sigma(Q)$ is $\ell\text{-}solo\}$ (see Section **??**).

We show that $S$ satisfies the two parts of Theorem **??**, as follows.

*Proof of Part (a):* Let $Q$ be any proper subset of $P_n$. We claim that the algorithm $\mathcal{B}'_n$ shown in Section **??** solves consensus in the subsystem $S(Q)$ of $S$. To prove this, we show that the algorithm $\mathcal{B}'_n$ satisfies the properties of consensus in every run of the processes in $Q$ where either (a) all processes in $Q$ crash (the corresponding schedule of $Q$ is finite), or (b) there is a process in $Q$ that executes at least $\ell$ consecutive steps solo infinitely often. In other words, the algorithm $\mathcal{B}'_n$ solves consensus in every run of the processes in $Q$ whose schedule is $\ell\text{-}solo$. By Observation **??**, every schedule of subsystem $S(Q)$ of system $S = S_n^\ell$ is an $\ell\text{-}solo$ schedule of $Q$, thus the algorithm $\mathcal{B}'_n$ solves consensus in $S(Q)$.

The detailed explanation and proof of algorithm $\mathcal{B}'_n$ is given in Section **??**.

*Proof of Part (b):* We claim that consensus cannot be solved in system $S$. Suppose, for contradiction, that there is an algorithm $\mathcal{C}_n$ that solves consensus in system $S$. Then we can show that consensus can be solved in the asynchronous system $A_2$ — contradicting the well-known impossibility result in [**?**, **?**]. To solve consensus in $A_2$, the two processes of $A_2$ can *simulate* the execution of the consensus algorithm $\mathcal{C}_n$ by the $n$ processes of $P_n$ in a system $S'$ such that:

(1) $S'$ is a subsystem of $S = S_n^\ell = [P_n, \Sigma_n^\ell]$. In particular, every schedule of $S'$ is in also a schedule of $S$.

(2) If at least one of the two processes of system $A_2$ does not crash during their simulation of algorithm $\mathcal{C}_n$ in system $S'$, then at least one of the $n$ simulated processes in $P_n$ does not crash in the simulated run of $\mathcal{C}_n$ in system $S'$.

Property (1), together with the assumption that $\mathcal{C}_n$ solves consensus in $S$, ensures that every simulated run of $\mathcal{C}_n$ in the subsystem $S'$ of $S$ also satisfies the properties of consensus. In particular, in each simulated run of $\mathcal{C}_n$ in $S'$, all the simulated processes that are correct in this run reach a common decision.

Property (2) ensures that if at least one of the two processes in $A_2$ is correct, then at least one of the $n$ processes in $P_n$ that they simulate, say process $p$, is also correct; by the above, $p$ reaches a decision in the simulated run of $\mathcal{C}_n$ in $S'$. So, every correct process in $A_2$ can wait until one of the simulated processes decides some value, and then adopt this common decision value.

Thus, by simulating the execution of $\mathcal{C}_n$ in system $S' \subseteq S$, the two processes of $A_2$ can solve consensus in $A_2$. This violation of the impossibility result of [**?**, **?**] concludes the proof that consensus cannot be solved in $S$.

In Section **??** we show how the two processes of system $A_2$ simulate the execution of the consensus algorithm $\mathcal{C}_n$ by the processes $P_n$ in system $S'$ such that conditions (1) and (2) above hold.

# 4 Consensus algorithm

In this section we present an algorithm $\mathcal{B}'_n$ in a system $S = [P, \Sigma]$ where $P$ is a set of $n$ processes. We show that for some $\ell$ it achieves uniform consensus in any $\ell\text{-}solo$ schedules. Moreover for any $Q \subset P$, $\mathcal{B}'_n$ solves the consensus in the subsystem $S_n^\ell(Q)$.

For convenience, we first give $\mathcal{B}_n$ an algorithm using multi-writer wait-free snapshot then we deduce from $\mathcal{B}_n$ the existence of an algorithm $\mathcal{B}'_n$ using only SWMR registers solving consensus in any $\ell\text{-}solo$ schedules.

Algorithm $\mathcal{B}_n$ (Figure **??**) is an adaptation of obstruction-free consensus algorithm in [**?**]. It uses a $2n$ multi-writer snapshot array and terminates if at least one process runs alone during $2n + 1$ iterations of the main loop. In algorithm $\mathcal{B}_n$ we say that process decides value $x$ if it executes Line **??** with variable $v$ equal to $x$.

Describe informally the algorithm. Each process maintains a proposed value (variable $propose$) and tries to write its proposed value in a cyclic order in every cell of the snapshot array $R$. The proposed value is modified

Shared variables:

$R[1..2n]$: `array of multi-writer snapshot`, initialized to $[\perp..\perp]$

CODE FOR PROCESS $p$:

```
1  prop := v_p                                                        /*p input value*/
2  View[1..2n] : array, initialized to [⊥..⊥]
3  decide := ⊥
4  i := 1
5  forever do
6      update(R[i], prop)
7      View := scan(R)
8      if ∃v : ∀j(1 ≤ j ≤ 2n)View[j] = v
9      then                                                           /* one value in R */
10         let v such that ∀j(1 ≤ j ≤ 2n) : v = View[j]
11         if decide == ⊥
12         then
13             decide := v
14     else                                                           /* more than one value in R */
15         if (∃v(v ≠ ⊥) :| {j | (1 ≤ j ≤ 2n) ∧ v = View[j]} |> n
16         then                                                       / * a majority of cells of R contain v */
17             let v (v ≠ ⊥) such that (| {j | ∧(1 ≤ j ≤ 2n)) ∧ v = View[j]} |> n
18             prop := v
19     i := (i mod 2n) + 1
```

Figure 1: $\mathcal{B}_n$ consensus algorithm with $2n$ multi-writer snapshot array.

following the following rules: initially it is the initial value of the process, then just after the process writes its proposed value in some cell of the snapshot array $R$, it scans $R$ and if some value $v$ is the value in a majority of cells of the returned array, the process adopts that value as proposed value. When all values in the snapshot array are the same, then the process decides this value.

If some process decides value $v$, in the last scan made by this process all the $2n$ cells of the snapshot array are equal to $v$ and then it can be proved that all the next scans of $R$ will have a strict majority of cells with value $v$. Hence the proposed value of any process that makes a scan after that will be $v$ and then $v$ is the only value that can be decided. Moreover if some process is running alone enough time it will be able to update all the cells of $R$ with its proposed value and then it decides. Consider the body of the main loop of algorithm $\mathcal{B}_n$ (Lines **??** to **??**), if a process runs alone at least $2n + 1$ successive iterations of the body of the main loop, after the scan of its first iteration its proposed value does not change and it updates during the next $2n$ iterations all cells of the snapshot array with this value. Then by the end of these $2n + 1$ iterations it decides. As soon as some process decides, it is easy to verify that all correct processes decide.

**Theorem 4** *If some process runs alone during $2n + 1$ successive complete iterations of the main loop, algorithm $\mathcal{B}_n$ is a consensus algorithm.*

PROOF. Due to the lack of space, we give the formal proof of this Theorem in Appendix (Section **??**). □

Algorithm $\mathcal{B}_n$ uses a multi-writer snapshot array. Using the implementation of multi-writer snapshot with MWMR registers described in [**?**] and the implementation of MWMR registers with SWMR registers from [**?**], we can derive from $\mathcal{B}_n$ an algorithm $\mathcal{B}'_n$ using only SWMR registers. From Theorem **??**, algorithm $\mathcal{B}'_n$ is a consensus algorithm in which all correct processes terminate in any schedule such that some process takes enough atomic steps solo to run alone during $2n + 1$ successive iterations of the main loop.

The algorithm in [**?**] gives a wait-free implementation of a multi-writer snapshot array from MWMR registers. For this algorithm there exists a constant $C_1$ such that each update or scan operation requires less than $C_1 n^2$ reads and writes to MWMR registers. Moreover, if the values written in the snapshot are bounded by some $K$ there exists a function $D_1$ such that the values written in the MWMR registers are bounded by $D_1(K)$.

The algorithm given in [**?**] implements MWMR registers from SWMR registers. For this algorithm there exists a constant $C_2$ such that each write requires less than $C_2 n$ reads and writes to the SWMR registers and each read requires less than $C_2 n \log n$ reads and writes to the SWMR registers. Moreover, if the values written in the MWMR registers are bounded by $K$ there exists a function $D_2$ the value written in the SWMR registers are bounded by $D_2(K)$

Let $m = (2n+2)(2C_1 n^2 C_2 n \log n)$, if some process runs solo for at least $m$ steps it runs alone during $2n+1$ iteration of the main loop of $\mathcal{B}_n$, then by Theorem **??** all correct processes decide.

In algorithm $\mathcal{B}_n$ the processes update the snapshot array with only initial values. Let $M$ be the max of the initial values of processes, in $\mathcal{B'}_n$ all SWMR are bounded by a constant namely by $D_2(D_1(M))$. Then we get:

**Theorem 5** *There is a constant $\ell$ such that algorithm $\mathcal{B}'_n$ solves consensus with a bounded number of bounded-size SWMR registers in any $\ell$-solo schedules.*

Consider system $S_n^\ell$ for the bound $\ell$ given in the previous theorem, then for any $Q \subset P_n$, by definitions of $S_n^\ell$, each schedule of $S_n^\ell(Q)$ is $\ell$-solo for some process $q \in Q$. Hence $\mathcal{B}'_n$ solves the consensus in $S_n^\ell(Q)$:

**Theorem 6** *For any proper subset $Q$ of $P_n$, consensus can be solved in $S_n^\ell(Q)$ by an algorithm with a bounded number of bounded-size SWMR registers.*

# 5 Consensus cannot be solved in system $S$

Shared variables:

/* Program Counters of simulated processes $1, 2, \ldots, n$ */

$PC[1..n]$: `array of SWSR registers`, initialized to [0..0]

CODE FOR PROCESS $x$: /* process $x$ simulates process 1 executing algorithm $\mathcal{C}_n$*/

1 input value of process 1 in $\mathcal{C}_n$ := input value of process $x$
2 **forever do**
3 $PC[1] := PC[1] + 1$
4 execute one step of process 1 running algorithm $\mathcal{C}_n$
5 **if** process 1 decides some value $v$ in $\mathcal{C}_n$ **then** decide $v$

CODE FOR PROCESS $y$: /* process $y$ simulates processes $2, 3, \ldots, n$ executing algorithm $\mathcal{C}_n$*/

Local variables:

$pc[1..n]$: `array`
$p$: `scalar`

6 **for** $p = 2$ **to** $n$ **do**
7 input value of process $p$ in $\mathcal{C}_n$ := input value of process $y$
8 **for** $i = 1, 2, \ldots$ **do** /* simulation of $\{1, p\}^\ell$ steps of processes 1 and $p$ */
9 $p := 2 + (i-1) \bmod (n-1)$ /* with $p = 2, 3, \ldots, n, 2, \ldots$ in round-robin order */
10 $pc[1] := PC[1]$
11 $pc[p] := PC[p]$
12 **while** $(PC[1] \leq pc[1] + \ell)$ **or** $(PC[p] < pc[p] + \ell)$ **do**
13 $PC[p] := PC[p] + 1$
14 execute one step of process $p$ running algorithm $\mathcal{C}_n$
15 **if** process $p$ decides a value $v$ in $\mathcal{C}_n$ **and** $y$ has not yet decided **then** decide $v$

Figure 2: Processes $x$ and $y$ simulate the execution of $\mathcal{C}_n$ by processes $1, 2, \ldots, n$ in system $S' \subseteq S = S_n^\ell$.

We now prove that consensus cannot be solved in system $S = S_n^\ell$. Suppose, for contradiction, that there is an algorithm $\mathcal{C}_n$ that solves consensus in system $S$. We show that the two processes of the asynchronous system $A_2$ can use $\mathcal{C}_n$ to solve consensus among themselves (contradicting the impossibility result in [**?**, **?**]); intuitively,

they do so by simulating the execution of $\mathcal{C}_n$ by the $n$ processes of $P_n$ in a subsystem $S'$ of $S$, such that if one of the processes in $A_2$ does not crash then at least one of simulated processes in $P_n$ does not crash. Henceforth, we denote by $x$ and $y$ the two processes of $A_2$ (this is to distinguish them from the $n$ processes of $P_n = \{1, 2, \ldots, n\}$ that they simulate).

In the following: (a) we first define the system $S'$ and show that it is indeed a subset of system $S$, (b) we then show how the algorithm in Figure **??** executed by the two processes $x$ and $y$ of $A_2$ simulates runs of $\mathcal{C}_n$ in system $S'$, i.e., the schedules of these runs are schedules of $S'$, and (c) finally we show that $x$ and $y$ solve consensus among themselves using this simulation algorithm.

Intuitively, the schedules of system $S'$ are: (1) all the infinite schedules of the form $\prod_{i=1}^{\infty}[\{1,2\}^\ell\{1,3\}^\ell \ldots \{1,n\}^\ell]$,[4] (2) all the finite prefixes of such schedules, and (3) all the finite prefixes of such schedules followed by $p^\infty$ for some process $p \in P_n$ (i.e, an infinite sequence of steps of $p$). More precisely, let $\Sigma = \{\sigma \mid \sigma$ is of the form $\prod_{i=1}^{\infty}[\{1,2\}^\ell\{1,3\}^\ell \ldots \{1,n\}^\ell]$.

**Definition 7** *$S' = [P_n, \overline{\Sigma_n^\ell}]$ where $\overline{\Sigma_n^\ell} = \{\sigma | \sigma \in \Sigma$ or there is a finite prefix $\sigma'$ of a schedule in $\Sigma$ such that $\sigma = \sigma'$ or such that $\sigma = \sigma' p^\infty$ for some $p \in P_n\}$.*

**Lemma 8** *$S'$ is a subsystem of system $S$.*

PROOF. Since $S' = [P_n, \overline{\Sigma_n^\ell}]$ and $S = [P_n, \Sigma_n^\ell]$, we must show that $\overline{\Sigma_n^\ell} \subseteq \Sigma_n^\ell$. Recall that $\Sigma_n^\ell = \{\sigma \mid \sigma \in \Sigma_n$ and for all $Q \subset P_n : \sigma(Q)$ is $\ell\text{-}solo\}$. Let $\sigma \in \overline{\Sigma_n^\ell}$. Since $\sigma$ is a schedule of $P_n$, $\sigma \in \Sigma_n$. To show that $\sigma \in \Sigma_n^\ell$ it suffices to prove that for all $Q \subset P_n$, $\sigma(Q)$ is $\ell\text{-}solo$. Let $Q$ any proper subset of $P_n$. If $Q$ is empty then $\sigma(Q)$ is trivially $\ell\text{-}solo$, so assume that $Q$ is not empty. There are three possible cases:

1. $\sigma$ is an infinite sequence of the form $\prod_{i=1}^{\infty}[\{1,2\}^\ell\{1,3\}^\ell \ldots \{1,n\}^\ell]$.

   Suppose process 1 is in $Q$. Since $Q$ is a proper subset of $P_n$, there is a process $p \in P_n \setminus Q$. Note that subsequences of the form $\{1,p\}^\ell$ appears infinitely often in $\sigma$. Thus, since $p \notin Q$, the subsequence $1^\ell$ appears infinitely often in $\sigma(Q)$. In other words, process 1 runs solo for $\ell$ steps infinitely often in $\sigma(Q)$. So $\sigma(Q)$ is $\ell\text{-}solo$.

   Suppose process 1 is not in $Q$. Since $Q$ is not empty, there is a process $p \in Q$. Note that subsequences of the form $\{1,p\}^\ell$ appears infinitely often in $\sigma$. Thus, since $1 \notin Q$, the subsequence $p^\ell$ appears infinitely often in $\sigma(Q)$. So $\sigma(Q)$ is $\ell\text{-}solo$.

2. $\sigma = \sigma'$ for some finite schedule $\sigma'$. Since $\sigma$ is finite it is trivially $\ell\text{-}solo$.

3. $\sigma = \sigma' p^\infty$ for some finite schedule $\sigma'$ and some process $p$. If $p \in Q$, then $\sigma(Q)$ is of the form $\sigma'' p^\infty$ for some $\sigma''$. Thus $p^\ell$ appears infinitely often in $\sigma(Q)$, and so $\sigma(Q)$ is $\ell\text{-}solo$. If $p \notin Q$ then $\sigma(Q)$ is finite, so it is trivially $\ell\text{-}solo$.

□

We now show that when processes $x$ and $y$ execute the algorithm in Figure **??** in the asynchronous system $A_2$, they simulate schedules of system $S'$. More precisely:

**Lemma 9** *When processes $x$ and $y$ execute the algorithm in Figure **??** in system $A_2$, they simulate runs of $\mathcal{C}_n$ by the processes $P_n$ in system $S'$, i.e., the schedules of these simulated runs are schedules of $S'$.*

PROOF. First note that each time process $x$ executes an iteration of its forever loop (lines **??**-**??**), it increments $PC[1]$ and does one step of process 1 executing algorithm $\mathcal{C}_n$. Similarly, each time process $y$ executes an iteration of its while loop (lines **??**- **??**) for a process $p \in \{2, \ldots, n\}$, it increments $PC[p]$ and does one step of process $p$ executing algorithm $\mathcal{C}_n$. Thus, it is clear that $x$ and $y$ simulate runs of $\mathcal{C}_n$ by the processes in $P_n$. It remains to

[4] Recall that $\{p,q\}^i$ is any sequence of steps of $p$ and $q$ that contains *at least* $i$ steps of $p$ and *at least* $i$ steps of $q$, in any order.

show that the schedules of these simulated runs are schedules of the system $S' = [P_n, \overline{\Sigma_n^\ell}]$, i.e., they are either (1) infinite schedules of the form $\prod_{i=1}^{\infty}[\{1,2\}^\ell\{1,3\}^\ell \ldots \{1,n\}^\ell]$, or (2) finite prefixes of such schedules, or (3) finite prefixes of such schedules followed by $p^\infty$ for some process $p \in P_n$.

From the above and the termination condition of processs $y$'s while loop of line **??**, it is clear that $y$ completes each execution of the while loop that it starts, unless it crashes or process $x$ crashes (and stops incrementing $PC[1]$). Thus, unless $x$ or $y$ crash, process $y$ executes an infinite number of iterations of the for-loop of line **??**. Note that during its $i$-th iteration of this for-loop, process $y$ simulates the steps of process $p = 2+(i-1)\text{mod}(n-1)$. So in the successive iterations of this for-loop, process $y$ simulates the steps of the processes $2, 3, ...., n$ in round-robin order.

Let $t_i$ be the time when process $y$ starts its $i$-th iteration of the for-loop of line **??**; $t_i$ is undefined if $y$ never starts this iteration. From the above, we have the following:

**Observation 10** *If, for some $k \geq 1$, $t_k$ is undefined then process $x$ or $y$ (or both) crash.*

To show that $x$ and $y$ simulate schedules of $S'$, we consider the steps of the processes in $P_n$ that $x$ and $y$ simulate from time 0 (when $x$ or $y$ start executing the simulation algorithm) to time $t_1$, from time $t_1$ to time $t_2$ , ..., from time $t_j$ to time $t_{j+1}$, ... until we reach a time $t_k$ that is undefined if such a time exists.

Note first that if $t_1$ is not defined, then $y$ crashes before executing its first for-loop of line **??**, so $y$ never simulates any step. Since process $x$ simulates only the steps of process 1, the resulting simulated schedule of $P_n$ is simply $1^\infty$ or some finite prefix of $1^\infty$ (if $x$ crashes). It is easy to see that this is a schedule of $S'$.

Henceforth assume that $t_1$ is defined. During the interval $[0, t_1]$ process $y$ does not simulate any step, and process $x$ simulates only steps of process 1. So during interval $[0, t_1]$ the simulated schedule is some finite prefix of $1^\infty$.

Now suppose that, for some $i \geq 1$, $t_i$ is defined. Let $p = 2 + (i-1)\text{mod}(n-1)$. As we noted before, this is the (only) process of $P_n$ that $y$ simulates during its $i$-th iteration of the for-loop of line **??** that starts at time $t_i$.

There are two possible cases:

(1) $t_{i+1}$ is defined. In this case, we show that during the interval of time $[t_i, t_{i+1}]$, processes $x$ and $y$ simulate a sequence of steps of the form $\{1,p\}^\ell$.

CLAIM 1: During the interval $[t_i, t_{i+1}]$ processes $x$ and $y$ simulate only the steps of processes 1 and $p$, and they simulate at least $\ell$ steps of 1 and at least $\ell$ steps of $p$.

*Proof of Claim 1:* First note that during interval $[t_i, t_{i+1}]$, process $y$ simulates only steps of process $p$, and process $x$ simulates only steps of process 1.

Process $y$ stores the value of $PC[1]$ in $pc[1]$ at some time $\tau_1$, and $y$ stores the value of $PC[p]$ in $pc[p]$ at some time $\tau_2$, such that $t_i \leq \tau_1 \leq \tau_2 < t_{i+1}$. Furthermore, the while loop that $y$ executes during the interval $[t_i, t_{i+1}]$ ends at some time $\tau_3 \leq t_{i+1}$, when $y$ finds that $(PC[1] > pc[1]+\ell)$ and $(PC[p] \geq pc[p]+\ell)$ holds. Since $PC(1) = pc(1)$ at time $\tau_1$ and $PC[1] > pc[1] + \ell$ at time $\tau_3$, then at least $\ell$ steps of process 1 are simulated during the interval $[\tau_1, \tau_3]$. Similarly, since $PC(p) = pc(p)$ at time $\tau_2$ and $PC[p] \geq pc[p] + \ell$ at time $\tau_3$, then at least $\ell$ steps of process $p$ are simulated during the interval $[\tau_2, \tau_3]$. We conclude that during interval $[t_i, t_{i+1}]$, only steps of processes 1 and $p$ are simulated, and at least $\ell$ steps of 1 and at least $\ell$ steps of $p$ are simulated.

(2) $t_{i+1}$ is undefined.

CLAIM 2: After time $t_i$, only the steps of processes 1 and $p$ are simulated. Furthermore, there is a time $\tau \geq t_i$ after which only steps of process 1 are simulated, or only steps of process $p$ are simulated, or no steps are simulated.

*Proof of Claim 2:* Since $t_{i+1}$ is undefined, process $y$ never starts its $(i+1)$-th iteration of the for-loop of line **??**. Thus, after time $t_i$ process $y$ can simulate only the steps of process $p$. Since $x$ simulates only the steps

of process 1, after time $t_i$ only the steps of 1 and $p$ can be simulated. Furthermore, since $t_{i+1}$ is undefined, by Observation **??** process $x$ or process $y$ (or both) crash. If $y$ crashes then after this crash occurs no steps of $p$ are simulated. If $x$ crashes then after this crash occurs no steps of 1 are simulated. So there is a time $\tau \geq t_i$ after which only steps of process 1 are simulated, or only steps of process $p$ are simulated, or no steps are simulated.

From the above, it is clear that when $x$ and $y$ execute the algorithm in Figure **??** in system $A_2$, they simulate a schedule of $P_n$ of the form $\prod_{i=1}^{\infty}[\{1,2\}^{\ell}\{1,3\}^{\ell}\ldots\{1,n\}^{\ell}]$, or a finite prefix of such a schedule, or a finite prefix of such a schedule followed by $p^{\infty}$ for some process $p \in P_n$. In other words, they simulate schedules of the system $S' = [P_n, \overline{\Sigma_n^{\ell}}]$. □

We now show that the algorithm Figure **??** solves consensus in the asynchronous system $A_2$. This algorithm simulates the execution of an algorithm $\mathcal{C}_n$ that is assumed to solve consensus in system $S$. We first show that $\mathcal{C}_n$ satisfies the *uniform* version of the agreement and integrity properties, namely: (a) all the processes that decide (whether correct or faulty) decide the same value, and (b) any process that decides (whether correct or faulty) decides a process input value. That is:

**Lemma 11** *In system $S = [P_n, \Sigma_n^{\ell}]$, the consensus algorithm $\mathcal{C}_n$ satisfies the uniform agreement and uniform integrity properties.*

PROOF. Due to the lack of space, we give the proof of this Lemma in Appendix (Sec **??**). □

To prove that the algorithm in Figure **??** solves consensus in $A_2$, consider an execution of this algorithm where $x$ and $y$ have input value $i_x$ and $i_y$, respectively. In this execution, process $x$ simulates the steps of process 1 executing algorithm $\mathcal{C}_n$ with input $i_x$ (see line **??**); if process 1 decides a value $v$ in $\mathcal{C}_n$, then $x$ also decides $v$. Similarly, process $y$ simulates the steps of processes $2, 3, \ldots, n$ executing algorithm $\mathcal{C}_n$ with input $i_y$ (see line **??**). If any process in $\{2, 3, \ldots, n\}$ decides a value in $\mathcal{C}_n$, then $y$ also decides this value. By Lemma **??**, this execution simulates a run of the consensus algorithm $\mathcal{C}_n$ among the $n$ processes of $P_n$ in the subsystem $S'$ of $S$. We now show that $x$ and $y$ reach consensus.

- *(Uniform) Agreement:* If $x$ and $y$ decide, then $x$ decides the value that process 1 decides and $y$ decides the value that some process $p \in \{2, 3, \ldots, n\}$ decides in the simulated execution of $\mathcal{C}_n$ in system $S' \subseteq S$. By Lemma **??**, $\mathcal{C}_n$ satisfies the uniform agreement property in system $S$. Thus, $x$ and $y$ decide the same value.

- *Termination:* If process $x$ is correct, then process 1 is correct (i.e., it takes an infinite number of steps) in the simulated execution of $\mathcal{C}_n$ in system $S' \subseteq S$. Since $\mathcal{C}_n$ satisfies the termination property in system $S$, correct process 1 decides a value in this execution of $\mathcal{C}_n$. So $x$ also decides a value.

  If process $y$ is correct, then at least one process $p \in \{2, 3, \ldots, n\}$ that $y$ simulates is correct in the simulated execution of $\mathcal{C}_n$ in system $S' \subseteq S$. Since $\mathcal{C}_n$ satisfies the termination property in system $S$, correct process $p$ decides a value in this execution of $\mathcal{C}_n$. So $y$ also decides a value.

- *(Uniform) Integrity:* If $x$ or $y$ decides a value $v$, then some process $p \in P_n$ decides $v$ in the simulated execution of $\mathcal{C}_n$ in system $S' \subseteq S$. By Lemma **??**, $\mathcal{C}_n$ satisfies the uniform integrity property in system $S$. Thus $v$ must be the input value of some process $q \in P_n$ in this execution of $\mathcal{C}_n$. Note that the input value of $q$ in $A_n$ is the input value of $x$ or $y$ (algorithm lines **??** and **??**). So $v$ is the input value of $x$ or $y$.

We proved that if an algorithm $\mathcal{C}_n$ solves consensus in system $S$, then the algorithm in Figure **??** solves (uniform) consensus in the asynchronous system of two processes $A_2$ — contradicting the results in [**?**, **?**]. Thus, consensus cannot be solved in $S$.

# Appendix

## A Consensus (Section ??)

More formally, let $V^\tau(v)$ be the number of cells of snapshot array $R$ equal to $v$ at time $\tau$. Given any value $v$, assume that $p$ has already made its scan in Line **??**, the next update of a process $p$ in Line **??** will be with a value different from $v$ if (i) at least $n$ cells of the array returned by the last scan of $p$ contain the same value $w$ and $w \neq v$ and (ii) not all the values of the array returned by the last scan of $p$ are the same. Let $C^\tau(v)$ be the number of processes that have already made the scan (Line **??**) and have not yet made the next update (Line **??**) for which conditions (i) and (ii) are true. Let $Inv^\tau(v) \equiv V^\tau(v) - C^\tau(v) > n$, we have:

**Lemma 12** *If at some time $\tau$ $Inv^\tau(v)$ is true then for all time $\tau' \geq \tau$, $Inv^{\tau'}(v)$ is true.*

PROOF. Prove that for all $v$ $Inv^\tau(v)$ is an inductive invariant. Assume that $Inv^\tau(v)$ is true and consider the next step of any process and let $\tau'$ be the linearization time of the this next step and $p$ be the process making this step. We have the following cases:

- at time $\tau'$ process $p$ updates cell $i$ with a value $w$. As $p$ has performed its update and has not preformed yet its scan, $C^{\tau'}(v) = C^\tau(v) - 1$. If $v = w$ then $V^{\tau'} \geq V^\tau$ else $V^{\tau'} \geq V^\tau - 1$, in both cases $Inv^{\tau'}$ is true.
- at time $\tau'$ process $p$ performs a scan, the snapshot returned is the value of $R$ at time $\tau$, hence the number of cells equal to $v$ is $V^\tau(v)$, and we have $V^{\tau'}(v) = V^\tau(v)$. If $V^\tau(v) = 2n$ process $p$ decides hence $C^{\tau'}(v) < n$ and $Inv^{\tau'}(v)$ is true, else $Inv^\tau(v)$ ensures that the number of cells equal to $v$ in the scan is strictly greater than $n$ then the proposed value after Line **??** for $p$ is $v$ hence $C^{\tau'}(v) = C^\tau(v) - 1$ and $Inv^{\tau'}(v)$ is true.

Hence $Inv^\tau(v)$ is an invariant and by induction we deduce the lemma. □

**Lemma 13** *If some process $p$ decides $v$, if $\tau$ is the linearization time of the last scan of the process $p$ before its decision, then $Inv^\tau(v)$ is true. Moreover for every time $\tau' \geq \tau$ $V^{\tau'}(v) > n$.*

PROOF. As $p$ decides value $v$ at time $\tau$ we have $V^\tau(v) = 2n$ and as $C^\tau(v) < n$, $Inv^\tau$ is true. By Lemma **??** for every time $\tau' \geq \tau$ $V^{\tau'}(v) > n + C^{\tau'}$ and then $V^{\tau'}(v) > n$. □

We deduce the uniform agreement property:

**Lemma 14** *(Uniform agreement) If process $p$ decides $v$ and process $q$ decides $v'$ at time $\tau_2$ then $v = v'$.*

PROOF. Assume $p$ decides $v$ at time $\tau_1$ and $q$ decides $w$ at time $\tau_2$ Without loss of generality assume that $\tau_1 < \tau_2$. By Lemma **??** at any time $\tau \geq \tau_2$, $V^\tau(v) > n$ and $V^\tau(w) > n$. As the snapshot array has $2n$ cells, if value $v$ and value $w$ are each in more than $n$ cells then $v = w$. □

Only initial values are written in the snapshot array then we have the uniform integrity property:

**Lemma 15** *(Uniform integrity) If some process decides $v$ then $v$ is the initial value of some process.*

Concerning termination:

**Lemma 16** *If some process decides $v$ at time $\tau$ then all correct processes decide.*

PROOF. By Lemma **??**, after time $\tau$ forever more than $n$ cells of the snapshot array contains value $v$, then there is a time $\tau'$ after which any update of the snapshot array is with value $v$. Moreover any cell containing value $v$ after time $\tau'$ contains value $v$ forever. If some correct process $q$ does not decide, it will perform an infinite number of updates of the snapshot array after time $\tau'$ with value $v$ and as update of cells are made in a cyclic order eventually all cells of the snapshot array will be forever equal to $v$, and $q$ decides —a contradiction. □

**Lemma 17** *If some process runs alone during $2n+1$ successive iterations of the main loop, it decides.*

PROOF. Consider the body of the main loop of algorithm $\mathcal{B}_n$ (Lines **??** to **??**), if a process runs alone at least $2n+1$ successive iterations of the body of the main loop, after the scan of its first iteration its proposed value does not change and it updates during the next $2n$ iterations every cell of the snapshot array with this value. Then by the end of these $2n+1$ iterations it decides. □

By Lemma **??** and **??**, if some process runs alone during $2n+1$ successive iterations of the main loop, termination are ensured by algorithm $\mathcal{B}_n$. Then with agreement (Lemma **??**) and integrity (Lemma **??**) we get:

**Theorem 18** *(Theorem **??**) If some process runs alone during $2n+1$ successive iterations of the main loop, algorithm $\mathcal{B}_n$ is a consensus algorithm.*

In fact outr algorithm ensures uniform consensus.

# B Consensus cannot be solved in system $S$ (Section ??)

**Lemma 19** *( Lemma **??**) In system $S = [P_n, \Sigma_n^\ell]$, the consensus algorithm $\mathcal{C}_n$ satisfies the uniform agreement and uniform integrity properties.*

PROOF. Suppose, for contradiction that the lemma does not hold. There are two cases:

- $\mathcal{C}_n$ violates the uniform agreement property in system $S$. So there is a run $R$ of $\mathcal{C}_n$ in system $S$, i.e., a run $R$ with some schedule $\sigma \in \Sigma_n^\ell$, such that some process $p$ decides a value $v_p$, another process $q$ decides a value $v_p \neq v_p$, and at least one of these two processes crashes (i.e., stops taking steps) after deciding. Recall that $\Sigma_n^\ell = \{\sigma \mid \sigma \in \Sigma_n \text{ and for all } Q \subset P_n : \sigma(Q) \text{ is } \ell\text{-}solo\}$.

  Let $\alpha$ be the prefix of the schedule $\sigma$ that includes all steps of $\sigma$ up to and including the steps where $p$ and $q$ decide. Consider the infinite schedule $\sigma' = \alpha p^\ell q^\ell p^\ell q^\ell p^\ell q^\ell \ldots$. Note that both $p$ and $q$ are correct in schedule $\sigma'$. We claim that $\sigma' \in \Sigma_n^\ell$. This holds because for each proper subset $Q$ of $P_n$, $\sigma'(Q)$ is $\ell$-$solo$: in fact, if $Q$ contains $p$ or $q$, then $\sigma'(Q)$ contains infinite instances of $p^\ell$ or $q^\ell$, so it is $\ell$-$solo$; and if $Q$ contains neither $p$ nor $q$, then $\sigma'(Q)$ is finite and so it is also $\ell$-$solo$.

  Now consider the run $R'$ of $\mathcal{C}_n$ with the schedule $\sigma' \in \Sigma_n^\ell$ (so $R'$ is a run in system $S$) and with the same processes inputs as in run $R$. Since the schedule $\sigma$ and $\sigma'$ of $R$ and $R'$ have the same initial prefix of steps $\alpha$, processes $p$ and $q$ behave in same way in $R$ and $R'$ up to and including their decision steps: so they decide different values $v_p$ and $v_q$ in $R'$ even though they are both correct processes in $R'$. This contradicts the assumption that $\mathcal{C}_n$ solve consensus in $S$.

- $\mathcal{C}_n$ violates the uniform integrity property in system $S$. So there is a run $R$ of $\mathcal{C}_n$ in system $S$, i.e., a run $R$ with some schedule $\sigma \in \Sigma_n^\ell$, such that some process $p$ decides a value $v$ that is *not* the input value of a process, and and $p$ crashes after deciding.

Let $\alpha$ be the prefix of the schedule $\sigma$ that includes all steps of $\sigma$ up to and including the step where $p$ decides. Consider the infinite schedule $\sigma' = \alpha p^{\infty}$. It is clear that $p$ is correct in schedule $\sigma'$ and $\sigma' \in \Sigma_n^{\ell}$.

Now consider the run $R'$ of $\mathcal{C}_n$ with the schedule $\sigma' \in \Sigma_n^{\ell}$ (so $R'$ is a run in system $S$) and with the same processes inputs as in run $R$. Since the schedule $\sigma$ and $\sigma'$ of $R$ and $R'$ have the same initial prefix of steps $\alpha$, process $p$ behaves in same way in $R$ and $R'$ up to and including its decision step: so $p$ decides $v$ in $R'$ even though $v$ is not the input value of a process in $R'$. This contradicts the assumption that $\mathcal{C}_n$ solve consensus in $S$.

□