

When Is Recoverable Consensus Harder Than Consensus?

Carole Delporte-Gallet

*Université Paris Cité, IRIF
Paris, France*

Panagiota Fatourou

*Université Paris Cité, LIPADE
Paris, France
FORTH ICS and University of Crete
Heraklion, Greece*

Hugues Fauconnier

*Université Paris Cité, IRIF
Paris, France*

Eric Ruppert

*York University
Toronto, Canada*

FORTH ICS TR 491 December 2023

Abstract

We study the ability of different shared object types to solve recoverable consensus using non-volatile shared memory in a system with crashes and recoveries. In particular, we compare the difficulty of solving recoverable consensus to the difficulty of solving the standard wait-free consensus problem in a system with halting failures. We focus on the model where individual processes may crash and recover and the large class of object types that are equipped with a read operation. We characterize the readable object types that can solve recoverable consensus among a given number of processes. Using this characterization, we show that the number of processes that can solve consensus using a readable type can be larger than the number of processes that can solve recoverable consensus using that type, but only slightly larger.

1 Introduction

Recoverable consensus can play a key role in the study of asynchronous systems with non-volatile shared memory where processes can crash and recover, just as the standard consensus problem plays a central role in the study of asynchronous systems where processes may halt. In this paper, our goal is to leverage extensive research on the solvability of the standard consensus problem in systems equipped with different types of shared objects to gain knowledge about recoverable consensus in systems with non-volatile memory.

We consider an asynchronous model of computation, where processes communicate with one another by accessing shared memory. In particular, we are interested in studying how concurrent algorithms can take advantage of recent advances in non-volatile main memory, which maintains its stored values even when its power supply is turned off. This allows for algorithms that can carry on with a computation when processes crash and recover. We consider a standard theoretical model [3, 20, 22, 21] for this setting, where each process's local memory is volatile, but shared memory is non-volatile, and processes may crash and recover individually in an asynchronous manner. After a process crashes, its local memory, including its programme counter, is reinitialized to its initial state when the process recovers. Process crashes do not affect the state of shared memory. At recovery time, the process begins to execute its code again from the beginning¹. We refer to the sequence of steps that a process takes between crashes as a *run* of its code.

¹Alternatively, it could execute a recovery function. Our results hold either way. We use the simpler assumption of re-starting upon recovery to prove our results.

The consensus problem, where each process gets an input and all processes must agree to output one of them, has been central to the study of shared-memory computation in asynchronous systems with process halting failures (but no recoveries). A shared object type is defined by a sequential specification, which specifies the set of possible states of the object, the operations that can be performed on it, and how the object changes state and returns a response when an operation is applied on it. Herlihy [25] defined the consensus number of a type T , denoted $cons(T)$, to be the maximum number of processes that can solve consensus using objects of type T and read/write registers, or ∞ if there is no such maximum. The classification of types according to their consensus number is called the *consensus hierarchy*. This classification is particularly meaningful because of Herlihy’s universality result: a type T can be used (with registers) to obtain wait-free implementations of *all* object types in a system of n processes if and only if $cons(T)$ is at least n .

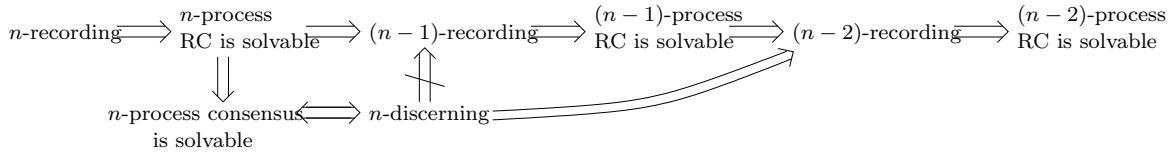


Figure 1: Relationships between conditions and solvability of consensus and recoverable consensus (with independent crashes) using a deterministic, readable type.

Golab [20] defined the *recoverable consensus* (RC) problem, where processes must agree on one of their input values, even if processes may crash and recover. An algorithm for RC defines a routine for each process to execute that takes an input value and eventually returns an output value, satisfying the following three properties.

- Agreement: no two output values produced are different. (This includes outputs by different processes and outputs of the same process when it performs multiple runs of the algorithm because it crashes and recovers.)
- Validity: each output value is the input value of some process.
- Recoverable wait-freedom: if a process executes its algorithm from the beginning, it either crashes or outputs a value after a finite number of its own steps.

Like Golab, we assume a process’s input value does not change, even across multiple runs, but this is not a crucial assumption. (If an RC algorithm requires this precondition, it can be transformed into one that does not using a register for each process’s input. When a process begins a run, it reads this register and, if it has not yet been written, the process writes its input value. It then uses the value in the register as its input, ensuring that all of the process’s runs of the original algorithm use the same input value.) Berryhill, Golab and Tripunitara [6] described how Herlihy’s universality result carries over to the model with crashes and recoveries, using RC in place of consensus. (See Section 4 for details.)

There are two common failure models for crashes and recoveries: *simultaneous crashes* [26], where all processes crash simultaneously, and *independent crashes* (introduced in [23] to study recoverable mutual exclusion), where processes can crash and recover individually in an asynchronous way. Golab [20] defined two recoverable consensus hierarchies. For an object type T , the *simultaneous RC number* of T is the maximum number of processes that can solve RC using an unbounded number of shared objects of type T and read/write registers when simultaneous crashes may occur. Similarly, the *independent RC number* of T , which we denote $rcons(T)$, is the maximum number of processes that can solve RC using shared objects of type T and read/write registers when independent crashes may occur. In both cases, if no maximum exists we

say the RC number is ∞ . This is a slight modification of Golab’s definition.² As an example, we show in Appendix H that $rcons(stack) = 1, rcons(stack) = 1$ [13], whereas it is known that $cons(stack) = 2$ [25].

1.1 Our Results

We focus on independent crashes since a simple extension of Golab’s result [20] described in Section 2 shows that RC has exactly the same difficulty as consensus in a system with simultaneous crashes.

Our main results are for deterministic shared object types that are *readable*, meaning that they are equipped with a read operation that returns the current state of the object without changing it. We define, for all $n \geq 2$, the n -recording property for shared object types. Roughly speaking, a readable type T is n -recording if n processes can be divided into two teams and use one object of type T to determine which of the two teams “wins”, even when processes crash and recover. The first team to perform an update operation on the object is the winning team, and this information is *recorded* in the object’s state, so that processes can determine which team wins by reading the object.

We show in Section 3.1 that being n -recording is sufficient for solving RC among n processes. We also show in Section 3.2 that the slightly weaker condition of being $(n - 1)$ -recording is necessary for solving RC among n processes. Thus, we have a fairly simple way of determining the approximate value of $rcons(T)$: if T is n -recording but not $(n + 1)$ -recording, we know that $rcons(T)$ is either n or $n + 1$.

Our n -recording property is related to Ruppert’s n -discerning property [33], which was defined to characterize readable types that can solve n -process consensus. In Section 3.3, we prove relationships between these two properties. This allows us to prove that if a type has consensus number n , then its RC number is between $n - 2$ and n . We give examples of types T with $rcons(T) = cons(T)$ and others with $rcons(T) < cons(T)$. In Section 3.4, we also use our characterization to show that weak types do not become much stronger (in terms of their power to solve RC) when used together. Section 4 describes how Herlihy’s motivation for studying the consensus hierarchy carries over to the RC hierarchy for the setting of non-volatile memory. See Figure 1 for an overview of our results.

2 Simultaneous Crash Model

In the case of simultaneous crashes, the RC hierarchy is identical to the standard consensus hierarchy.

Theorem 1. *Recoverable consensus is solvable among n processes using objects of type T and registers in the simultaneous crash model if and only if $cons(T) \geq n$.*

Golab [20] showed how to transform a standard consensus algorithm into an algorithm for RC in the case of simultaneous crashes. His transformation required a bound on the number of crashes to ensure that the space used by the algorithm is bounded. Since we allow an unbounded number of objects to be used to solve RC, a simple modification of Golab’s algorithm can be used to prove Theorem 1. See Appendix A for details. See the full version [13] for details. In

²Golab’s definition of RC numbers required the RC algorithms to use a *bounded* number of objects. We permit an infinite number of objects. When Jayanti [27] formalized Herlihy’s consensus hierarchy, he similarly allowed an unbounded number of objects to be used in solving consensus. (However, it follows from König’s Lemma [28] that any wait-free algorithm for the standard consensus problem that uses objects with finite non-determinism will use finitely many objects.) Universal constructions, which are one of the main motivations for studying the hierarchy, require an infinite number of instances of consensus anyway, so even if each instance uses a finite number of objects, the overall construction would still use an infinite number.

view of Theorem 1, we focus on determining RC numbers of types in the presence of *independent* crashes in the remainder of the paper.

3 Readable Objects

A *deterministic* object type has a sequential specification that specifies a unique response and state transition when a given operation is applied to an object of this type that is in a given state. An object is *readable* if it has a READ operation that returns the entire state of the object without altering it.³ Ruppert [33] provided a characterization of deterministic, readable types that can solve consensus among n processes. In this section, we develop a similar characterization for RC with independent crashes, and use this to compare the ability of types to solve the two problems.

The characterizations for consensus and for RC are linked to the *team consensus* problem, which is the problem of solving consensus when the set of processes are divided in advance into two non-empty teams and all processes on the same team get the same input. (This problem is also known as static consensus [31].)

We first review the characterization for standard consensus [33]. Suppose each process can perform a single update operation on an object O of type T , and then read O at some later time, and, based only on the responses of these two steps, determine which team updated O first. If this is possible, we say T is n -discerning.

Definition 2. *A deterministic type T is called n -discerning if there exist*

- a state q_0 ,
- a partition of n processes p_1, \dots, p_n into two non-empty teams A and B , and
- operations op_1, op_2, \dots, op_n

such that, for all $j \in \{1, \dots, n\}$, $R_{A,j} \cap R_{B,j} = \emptyset$, where $R_{X,j}$ is the set of pairs (r, q) for which there exist distinct process indices i_1, \dots, i_α including j with $p_{i_1} \in X$ such that if $op_{i_1}, \dots, op_{i_\alpha}$ are performed in this order on an object of type T initially in state q_0 , then op_j returns r and the object ends up in state q .

In this definition and in Definition 4, an operation op_i includes the name of the operation and any arguments to it. For example, WRITE(42) is an operation on a read/write register. Operations op_1, \dots, op_n need not be distinct. Ruppert used a valency argument to show that any deterministic, readable type that can solve consensus among n processes must be n -discerning. Conversely, team consensus can be solved using a readable n -discerning object O and one register per team as follows. Each process p_i writes its input in its team's register, performs its operation op_i on O and then reads O 's state. The process determines which team updated O first and outputs the value in that team's register. A tournament then solves consensus: processes within each team agree on an input value recursively and then run team consensus to choose the final output value. The argument sketched here yields the following characterization.

Theorem 3 ([33]). *A deterministic, readable type can be used, together with registers, to solve n -process wait-free consensus if and only if it is n -discerning.*

We now consider how to characterize readable types that can solve *recoverable* consensus, with independent process crashes. *Recoverable team consensus* is the RC problem where the processes are partitioned in advance into two non-empty teams and inputs are constrained so

³We use this definition for simplicity, but our results would apply equally well to the original, more general definition of readable objects in [33], which allows the state of the object to be read piece-by-piece. For example, an array of registers is also readable under the more general definition.

that all processes on the same team have the same input value. We shall show that RC is solvable if and only if recoverable team consensus is solvable: the only if direction is trivial, and the converse will be shown using the same tournament algorithm outlined above. So, it suffices to characterize types that can solve recoverable team consensus for n processes.

We shall define a property called n -recording such that a type T satisfying the property will allow n processes to solve recoverable team consensus in a simple way. A shared object O of type T is initialized to some state q_0 . To solve team consensus using an n -discerning type, each process performs a single operation on O and then reads O , and is able to conclude from the responses to these two steps which team updated O first. There are two key difficulties when we consider processes that may crash and recover: (1) if a process crashes after performing its update, thereby losing the response of that update, the process cannot use the response to determine which team won, and (2) a process that recovers should try to avoid performing its update on O a second time so that it does not obliterate the evidence of which team updated O first.

To cope with (1), our new n -recording property should allow a process to determine which team updated O first based simply on the state of O , which can be read at any time. Thus, two sequences of update operations that start with processes on opposite teams must not take O to the same state. This is formalized in condition 1 of Definition 4, below.

We now consider how to cope with (2). If O could never return to its initial state q_0 , checking that O 's state is q_0 before updating O would ensure that no process ever updates O twice. (See the code for processes on team A in Figure 2.) However, we can solve team consensus under a weaker condition: O 's state *can* return to q_0 after a process from team A updates O first, *provided that* team B has only one process. In this case, condition 1 of Definition 4 implies that the state cannot return to q_0 if a process on team B updates O first. Processes on team A behave as before, updating O if they find it in state q_0 . If $|B| > 1$, processes on team B do likewise. However, if $|B| = 1$, the lone process on team B updates O if it finds O in state q_0 *and* sees that no process on team A has started its algorithm: in this case it knows that no operation has been performed on O , since O can return to q_0 only if a process on team A updated it first. If the lone process on team B finds that a process on team A has already started, it simply outputs team A 's input value. (See the code for processes on team B in Figure 2.) This motivates condition 2 of Definition 4 below. A symmetric scenario motivates condition 3.

The approach of having processes on team B defer to team A if they see that a process on team A has started running works only if $|B| = 1$: if the algorithm used this approach with $|B| > 1$, one process on team B might start running before any process on team A and later go on to be the first process to update O , while another process on team B might start after a process on team A has taken steps and defer to team A . In this case, the latter process on team B would conclude that team A won, while others would conclude that team B won, violating agreement.

These considerations lead us to formulate the n -recording property in Definition 4, which uses the following notation. Fix a deterministic, readable type T . Let X be a subset of the set of all processes $\{p_1, \dots, p_n\}$ and let op_1, \dots, op_n be operations. Let q_0 be a state of type T . Define $Q_X(q_0, op_1, \dots, op_n)$ to be the set of all states q for which there exist distinct process indices i_1, \dots, i_α with $p_{i_1} \in X$ such that the sequence of operations $op_{i_1}, \dots, op_{i_\alpha}$ applied to an object of type T initially in state q_0 leaves the object in state q . We omit the parameters of Q_X when they are clear from context.

Definition 4. *A deterministic type T is n -recording if there exist*

- *a state q_0 ,*
- *a partition of n processes p_1, \dots, p_n into two non-empty teams A and B , and*

- operations op_1, \dots, op_n

satisfying the following three conditions.

1. $Q_A(q_0, op_1, \dots, op_n) \cap Q_B(q_0, op_1, \dots, op_n) = \emptyset$.
2. $q_0 \notin Q_A(q_0, op_1, \dots, op_n)$ or $|B| = 1$.
3. $q_0 \notin Q_B(q_0, op_1, \dots, op_n)$ or $|A| = 1$.

We call a type that satisfies this property n -recording because it *records* in its state information about the team that first updates the object, if it is initialized to state q_0 .

We first prove some simple consequences of Definition 4.

Observation 5. *For $n \geq 2$, if a deterministic type is n -recording, then it is n -discerning.*

To see why this is true, we can use the same choice of $A, B, q_0, op_1, \dots, op_n$ for both definitions. If, for some j , there were an $(r, q) \in R_{A,j} \cap R_{B,j}$ then q would also be in $Q_A \cap Q_B$, which would violate property 1 of the definition of n -recording. So we can conclude that $R_{A,j} \cap R_{B,j}$ must be empty, as required for the definition of n -discerning.

Observation 6. *For $n \geq 3$, if a deterministic type is n -recording, then it is $(n - 1)$ -recording.*

If a type satisfies the definition of n -recording with teams A and B , we can omit one process from the larger team to get a division of $n - 1$ processes into non-empty teams A' and B' . We use the same initial state q_0 and assign the same operation to each process to satisfy the definition $(n - 1)$ -recording.

We now summarize the results about deterministic, readable types that we prove in the remainder of this section. Theorem 8 shows that any readable type that is n -recording is capable of solving RC among n processes. We prove in Theorem 14 that all types that can solve RC among n processes satisfy the $(n - 1)$ -recording property. (This is true even if the type is not readable.) Given a specification of a shared type, it is fairly straightforward to check whether it is n -recording. By determining the maximum n for which a given readable object type T is n -recording, we can conclude that $rcons(T)$ is either n or $n + 1$.

We also prove that an n -discerning type must be $(n - 2)$ -recording (Theorem 16), but not necessarily $(n - 1)$ -recording (Proposition 19). As a corollary of these results, we show that $cons(T) - 2 \leq rcons(T) \leq cons(T)$. These relationships are summarized in Figure 1. In Theorem 22, we also show how the power of a collection of readable types to solve RC is related to the power of each type when used in isolation.

3.1 Sufficient Condition

We use the algorithm in Figure 2 to show that recoverable team consensus can be solved using a deterministic, readable object O whose type is n -recording. The intuition for the algorithm has already been described above, but we now describe the code in more detail. The code assumes $q_0 \notin Q_B$; if $q_0 \in Q_B$, then $q_0 \notin Q_A$ and we would reverse the roles of A and B in the code. Each process first writes its input in its team's register. It then reads O . If O is not in the initial state q_0 , then the process determines which team went first based on the state of O and returns the value written in that team's register (lines 11–12 and lines 27–28). Otherwise, it updates O before reading the state again (lines 8–9 and 23–24) to determine which team updated O first. There is one exception: if team B has only one process, it yields to team A (line 21) if it sees that some process on team A has already written its input value. This allows for the case where $q_0 \in Q_A$ and $|B| = 1$: it could be that a process on team A updated O first, and then other processes (including the process on team B , in a previous run) performed updates that returned O to state q_0 . In this case, those processes would have output team A 's input value,

```

1  shared variables
2      Object  $O$  of type  $T$ , initially in state  $q_0$ 
3      Registers  $R_A$  and  $R_B$ , initially in state  $\perp$ 

4  DECIDE( $v$ ) // code for process  $p_i$  on team  $A$ 
5       $R_A \leftarrow v$ 
6       $q \leftarrow O$ 
7      if  $q = q_0$  then
8          apply  $op_i$  to  $O$ 
9           $q \leftarrow O$ 
10     end if
11     if  $q \in Q_A$  then return  $R_A$ 
12     else return  $R_B$ 
13     end if
14 end DECIDE

15

16 DECIDE( $v$ ) // code for process  $p_i$  on team  $B$ 
17      $R_B \leftarrow v$ 
18      $q \leftarrow O$ 
19     if  $q = q_0$  then
20         if  $|B| = 1$  and  $R_A \neq \perp$  then
21             return  $R_A$ 
22         else
23             apply  $op_i$  to  $O$ 
24              $q \leftarrow O$ 
25         end if
26     end if
27     if  $q \in Q_A$  then return  $R_A$ 
28     else return  $R_B$ 
29     end if
30 end DECIDE

```

Figure 2: Algorithm for recoverable team consensus (assuming $q_0 \notin Q_B$).

so we must ensure that the process on team B does not perform its update again, since that could cause processes to output team B 's input value, violating agreement.

The next lemma will help us argue that the algorithm behaves correctly in the tricky case where $q_0 \in Q_A$ and $|B| = 1$.

Lemma 7. *Suppose $q_0, A, B, op_1, \dots, op_n$ satisfy the definition of n -recording for a deterministic type T . Let $X \in \{A, B\}$. If $q_0 \notin Q_X$ and i_1, \dots, i_α is a sequence of distinct process indices such that the sequence of operations $op_{i_1}, \dots, op_{i_\alpha}$ takes an object of type T from state q_0 to state q_0 , then the indices of all processes of team X appear in the sequence.*

Proof. To derive a contradiction, suppose the claim is false, i.e., $j \notin \{i_1, \dots, i_\alpha\}$ for some process p_j on team X . If p_{i_1} were on team X , then the fact that the sequence of operations $op_{i_1}, \dots, op_{i_\alpha}$ take the state of an object from q_0 to q_0 would imply that $q_0 \in Q_X$, contrary to our assumption. Thus, p_{i_1} must be on the opposite team \bar{X} . Let q_j be the state that results when op_j is applied to an object in state q_0 . We have $q_j \in Q_X$ since the sequence op_j takes an object from state q_0 to q_j . We also have $q_j \in Q_{\bar{X}}$ since the sequence $op_{i_1}, \dots, op_{i_\alpha}, op_j$ takes an object of type T from state q_0 back to state q_0 and then to state q_j . Thus, $q_j \in Q_X \cap Q_{\bar{X}}$, which violates condition 1 in the definition of n -recording. \square

To gain some intuition, we describe why the following bad scenario cannot occur when $|B| = 1$ and $q_0 \in Q_A$. Suppose a process p_1 on team B begins, sees $R_A = \perp$, and is poised to update O at line 23. Then, a process p_2 on team A runs to completion, updating O and deciding R_A . Then, other processes update O , returning O 's state to q_0 . If p_1 were still poised to update O at line 23, then it would decide R_B , violating agreement. But this cannot happen: Lemma 7 ensures that p_1 must have been among the processes that already applied their operations on O to return O 's state to q_0 .

We also describe why the condition $|B| = 1$ on line 20 is necessary. If this test were missing, consider an execution where one process p_1 on team B begins, sees $R_A = \perp$ and is about to update O at line 23. Then, a process p_2 on team A writes to R_A . Next, another process p_3 on team B sees that $R_A \neq \perp$ and decides R_A (at line 21). Finally, process p_1 resumes and updates O . Since it is the first process to update O , O 's state would then be in Q_B , so p_1 would then read O and decide R_B , violating agreement. We avoid this scenario by the test $|B| = 1$ of line 20: line 21 is executed only if B contains just one process (whereas two processes on team B are needed for the bad scenario described above).

Theorem 8. *If a deterministic, readable type T is n -recording, then objects of type T , together with registers, can be used to solve recoverable consensus for n processes.*

Proof. If team recoverable consensus can be solved, then RC can be solved. Processes on each team agree recursively on an input value for their team, and then use team consensus to determine the final output. See Appendix B the full version [13] for details.

Thus, it suffices to show that the algorithm in Figure 2 solves recoverable team consensus using a type T that satisfies the condition of the theorem. Since $Q_A \cap Q_B = \emptyset$, we know that either $q_0 \notin Q_A$ or $q_0 \notin Q_B$. Without loss of generality, assume $q_0 \notin Q_B$. (If this is not the case, just swap the names of the two teams.)

Recoverable wait-freedom is clearly satisfied, since there are no loops in the code. It remains to show that every execution of the algorithm satisfies validity and agreement.

Lemma 9. *Validity and agreement are satisfied in executions where no process ever performs an update on O .*

Proof. In this case, O remains in state q_0 forever. Thus, no process can reach line 11 or 27, since it would first have to update O at line 8 or 23, respectively. So, processes output only at line 21. By the test on line 20, R_A is written before a process outputs its value on line 21. Thus, all outputs are the input value of team A . \square

For the remainder of the proof of the theorem, consider executions where at least one update is performed on O . Let s be the first step in the execution that performs an update on O .

Lemma 10. *For $X \in \{A, B\}$, if a process on team X performs s and $q_0 \notin Q_X$, then O 's state is in Q_X at all times after s .*

Proof. We first show that no process performs more than one update on O . To derive a contradiction, suppose some process performs two updates on O . Let s' be the first step in the execution when a process performs its *second* update on O and let p_i be the process that performs s' . Let r' be p_i 's run of the code that performs s' . Since r' begins after p_i 's first update on O , r' begins after s . By definition of s' , each process does at most one update on O before s' . Thus, the state of O is in Q_X at all times between s and s' . Since $q_0 \notin Q_X$, the state of O is never q_0 between s and s' . This contradicts the fact that r' must read the state of O to be q_0 between s and s' ; otherwise r' would not perform s' .

Thus, each process performs at most one update on O . By the definition of Q_X , the state of O is in Q_X at all times after s . \square

We next prove a similar lemma for the case where $q_0 \in Q_A$. In this case, the situation is a little more complicated. The state of O might return to q_0 . If this happens, we show that each process updates O at most once before the state returns to q_0 , and that only processes of team A can update O after the state returns to q_0 and each process does so at most once. This is enough to ensure that O 's state remains in Q_A at all times.

Lemma 11. *If s is performed by a process of team A and $q_0 \in Q_A$, then O 's state is in Q_A at all times.*

Proof. Since $q_0 \in Q_A$, there is a unique process p_j on team B , by condition 2 of the definition of n -recording. O 's state is $q_0 \in Q_A$ at all times before s . It remains to show that O 's state is in Q_A at all times after s . We consider two cases.

First, suppose O is never in state q_0 after s . Consider any process p_i that performs an update on O . Let s_i be p_i 's first update on O . By definition, s_i is either equal to s or after s . Any run by p_i that begins after s_i (and hence after s) that reads O on line 6 or 18 sees a value different from q_0 , so it does not perform an update on O . Thus, no process performs more than one update on O . It follows from the definition of Q_A that O 's state is in Q_A at all times after s .

Now, suppose O 's state is equal to q_0 at some time after s . Let s'' be the first step at or after s that changes O 's state back to q_0 . We next prove that no process performs two updates on O between s and s'' (inclusive). To derive a contradiction, suppose some process performs two such updates. Let s' be the first step when any process performs its *second* update on O . By definition, s' is between s and s'' (inclusive). Let p_i be the process that performs s' and let r' be the run by p_i that performs s' . Since r' begins after p_i 's first update to O , r' begins after s . Thus, r' reads O 's state to be different from q_0 at line 6 or 18, and therefore fails the test on line 7 or 19. This contradicts the fact that r' updates O . Hence, each process performs at most one update on O between s and s'' (inclusive).

It follows from the definition of Q_A that the state of O is in Q_A at all times between s and s'' . By Lemma 7, the unique process p_j on team B updates O between s and s'' (inclusive).

Next, we argue that the process p_j on team B updates O exactly once in the entire execution. We have already seen that p_j updates O exactly once between s and s'' (inclusive). Any run by process p_j that begins after that first update to O by p_j (and therefore after s) would see that $R_A \neq \perp$, since the process on team A that performs s writes to R_A before s . That run by p_j would therefore pass the test on line 20 and could not update O on line 23.

Thus, any updates to O after s'' are by processes in A . If there are no updates to O after s'' , then O remains in state $q_0 \in Q_A$ at all times after s'' . If there is some update to O after s'' ,

let s''' be the first one. Since $q_0 \notin Q_B$ and no process on team B updates O after s'' , the state of O can never be q_0 after s''' , by Lemma 7. Consider any process p_i on team A that performs an update on O after s'' . Let s_i be p_i 's first update on O after s'' . By the definition of s''' , s_i is either s''' or after s''' . Any run by p_i that begins after s_i (and therefore after s''') that reads O on line 6 will see a value different from q_0 , so it does not perform an update on O . Thus, no process performs more than one update on O after s'' . It follows from the definition of Q_A that O 's state is in Q_A at all times after s'' . \square

Lemma 12. *Any output produced by a process on team A is the input value of the team that first updated O .*

Proof. Consider a run r of the code by a process in A that produces an output. If r reads O at line 6 before s , then it will read the value q_0 and read O again at line 9, which is after s . Thus, the value tested at line 11 is read from O after s .

If the first update to O is by a process on team A , the value tested is in Q_A , by Lemma 10 and 11. So, r outputs the value of R_A .

If the first update to O is by a process on team B , the value tested is in Q_B , by Lemma 10 and the fact that $q_0 \notin Q_B$. Since $Q_A \cap Q_B = \emptyset$, the value tested will not be in Q_A . So, r outputs the value of R_B .

In both cases, the relevant register is written before s , so r outputs the input value of the team that first updates O . \square

Lemma 13. *Any output produced by a process on team B is the input value of the team that first updated O .*

Proof. Consider any run r of the code by a process in B that produces an output. We consider three cases.

Case 1: a process from team A performs s .

We first show r returns a value read from R_A by considering two subcases.

- (a) $q_0 \in Q_A$. In this case $|B| = 1$, by condition 2 of the definition of n -recording. By Lemma 11, O 's state is in Q_A at all times, so r cannot return at line 28. Therefore, r outputs the value it reads from R_A at line 21 or 27.
- (b) $q_0 \notin Q_A$. By Lemma 10, O 's state is in Q_A at all times after s . If r reads O at line 18 before s , it will see q_0 and execute the test at line 20. Then, it will either return the value in R_A at line 21, or read O again at line 24 after s , getting a value in Q_A and returning the value in R_A at line 27.

To derive a contradiction, suppose R_A is still \perp when r reads it at line 21 or 27. Then, r returns before s , since R_A must be written before s . So r must have read q_0 from O at line 18. Thus, the test at line 19 is true and the test at line 20 is false, so r performs an update on O before s , contradicting the definition of s .

Therefore, r outputs team A 's input value, as required.

Case 2: A process from team B performs s and $|B| > 1$. Since $q_0 \notin Q_B$, it follows from Lemma 10 that O 's state is in Q_B at all times after s . If r reads O at line 18 before s , it will see q_0 and execute the test at line 20, which fails because $|B| > 1$. Then, it will read O again at line 24 after s , getting a value in Q_B and return the value in R_B at line 28. Since r wrote R_B at line 17, r outputs team B 's input value, as required.

Case 3: A process from team B performs s and $|B| = 1$. Let p_j be the unique process on team B . By Lemma 10 and the fact that $q_0 \notin B$, the state of O is in Q_B at all times after s .

If r is the run of p_j that performs s , then r sees $R_A = \perp$ on line 20; otherwise it would not execute line 23. So, if r returns a value, it reads O at line 24 after s and gets a value in Q_B . It must then return a value at line 28.

Any run r of p_j that ends before s evaluates the test at line 19 to true and the test at line 20 to false, so it must crash before reaching line 23 and does not produce an output.

If r is a run of p_j that starts after s , it reads a value in Q_B at line 18. Since $q_0 \notin Q_B$, it would return at line 28.

Thus, all outputs by p_j are read from R_B at line 28, which contains team B 's input value written at line 17. □

Lemmas 12 and 13 prove validity and agreement when some process updates O , completing the proof of Theorem 8. □

3.2 Necessary Condition

In this section, we show that being $(n - 1)$ -recording is a necessary condition for a deterministic type to be capable of solving n -process RC. This result holds whether the type is readable or not. The proof uses a valency argument [17]. Assuming an algorithm exists, the valency argument constructs an infinite execution in which no process ever returns a value. Unfortunately, in the case of RC, it is possible to have an infinite execution where no process returns a value (if infinitely many crashes occur). Thus, the proof considers a restricted set of executions where each execution must produce an output value for some process within a finite number of steps, and uses this restricted set to define valency. This technique was used by Golab [20] to prove a necessary condition (weaker than the 2-recording property) for solving 2-process RC. Lo and Hadzilacos [30] had previously used a similar technique of defining valency using a pruned execution tree. Attiya, Ben-Baruch and Hendler [3] also used a valency argument in the context of non-volatile memory in their proof that a recoverable test-and-set object cannot be built from ordinary test-and-set objects (and registers).

Theorem 14. *For $n \geq 3$, if a deterministic type T can be used, together with registers, to solve recoverable consensus among n processes, then T is $(n - 1)$ -recording.*

Proof. Assume there is an algorithm A for RC among n processes p_1, \dots, p_n using objects of type T and registers. Let \mathcal{E}_A be the set of all executions of A where p_2, \dots, p_n never crash, and in any prefix of the execution, the number of crashes of p_1 is less than or equal to the total number of steps of p_2, \dots, p_n .

Consider a finite execution γ in \mathcal{E}_A . Define γ to be *v-valent* if there is no decision different from v in any extension of γ in \mathcal{E}_A . An execution γ cannot be both *v-valent* and *v'-valent* if $v \neq v'$, since a failure-free extension of γ must eventually produce a decision. We call γ *univalent* if it is *v-valent* for some v , or *multivalent* otherwise.

To see that a multivalent execution exists, consider an execution with no steps where processes p_1 and p_2 have inputs 0 and 1. If p_1 runs by itself, it must output 0; if p_2 runs by itself it must output 1.

Next, we argue that there is a *critical execution* γ , i.e., a multivalent execution in \mathcal{E}_A such that every extension of γ in \mathcal{E}_A is univalent. If there were not, we could construct an infinite execution of \mathcal{E}_A in which every prefix is multivalent, meaning that no process ever returns a value. Such an execution could be constructed inductively by starting with a multivalent execution and, at each step of the induction, extending it to a longer multivalent execution.

This would violate the termination property of RC, since some process takes an infinite number of steps without crashing.

For $1 \leq i \leq n$, let v_i be the value such that γ followed by the next step of p_i 's algorithm is v_i -valent. We show not all of v_1, \dots, v_n are the same. To derive a contradiction, suppose they are all equal. Since γ is multivalent, some extension of γ in \mathcal{E}_A is v' -valent for some $v' \neq v_2$. By assumption, the next step of each process's algorithm produces a v_2 -valent execution, so the v' -valent extension must begin with a crash of p_1 . But the extensions of γ shown in Figure 3(a) are indistinguishable to p_2 . Thus, p_2 returns the same value in both, contradicting the fact that one extends a v_2 -valent execution and the other extends a v' -valent execution, where $v_2 \neq v'$.

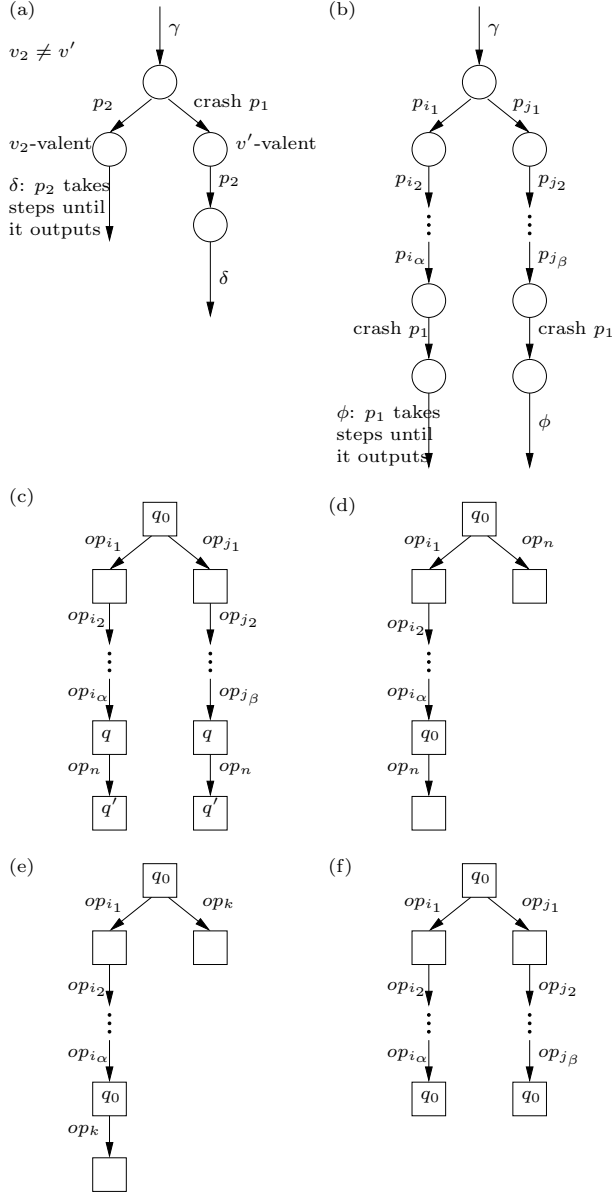


Figure 3: Proof of Theorem 14. Circles represent states of the system. Squares represent the state of O .

A standard argument shows that at the end of γ , each process is about to perform an operation on the same object O of type T , and that step cannot be a read operation. For $i \in \{1, \dots, n\}$, let op_i be the update operation that p_i is poised to perform on O after γ . Let q_0 be the state of O at the end of γ .

We next prove a technical lemma that will be used several times to complete the theorem's

proof. It captures a valency argument we use: if two sequences of steps by distinct processes chosen from p_1, \dots, p_n after γ can take O to the same state *and* process p_1 can crash after both of them, then the two extensions must have the same valency. To ensure that p_1 can crash, the hypothesis of the lemma requires that neither sequence consists of a single step by p_1 .

Lemma 15. *Suppose there is a sequence of distinct process ids i_1, \dots, i_α and another sequence of distinct ids j_1, \dots, j_β such that each sequence contains an element of $\{2, \dots, n\}$ and the sequences of operations $op_{i_1}, \dots, op_{i_\alpha}$ and $op_{j_1}, \dots, op_{j_\beta}$ both take object O from state q_0 to the same state q . Then, $v_{i_1} = v_{j_1}$.*

Proof. The two executions in Figure 3(b) are in \mathcal{E}_A since one of p_2, \dots, p_n takes a step in each extension of γ before p_1 crashes. O is in state q before p_1 crashes in both extensions, and no other shared object changes between the end of γ and the crash of p_1 . Thus, these two extensions are indistinguishable to the last run ϕ of A by p_1 . Since the left extension is v_{i_1} -valent and the right extension is v_{j_1} -valent, we must have $v_{i_1} = v_{j_1}$. \square

We now describe how to split $n - 1$ of the processes into two teams A and B according to their valency to satisfy the definition of $(n - 1)$ -recording. The following two cases describe how to relabel the processes (if necessary) so that we can split processes p_1, \dots, p_{n-1} into the two required teams.

Case 1: Suppose there is an i such that, for all $j \neq i$, $v_i \neq v_j$. Without loss of generality, assume that $i < n$. (If $i = n$, we can swap the ids of p_2 and p_n to ensure $i < n$, since $n \geq 3$.) Let $A = \{p_i\}$ and $B = \{p_1, \dots, p_{n-1}\} - \{p_i\}$.

Case 2: Suppose that for every i , there is a $j \neq i$ such that $v_i = v_j$. If there is a sequence of distinct ids i_1, \dots, i_α chosen from $\{1, \dots, n\}$ such that the sequence of operations $op_{i_1}, \dots, op_{i_\alpha}$ take the object O from state q_0 back to q_0 , then let $\ell = i_1$. Otherwise, let ℓ be any id. Without loss of generality, assume $\ell < n$. (If this is not the case, swap the labels of processes $n - 1$ and n to make it true.) Again, without loss of generality, assume $v_n \neq v_\ell$. (Since not all of v_1, \dots, v_n are the same, there is some ℓ' such that $v_{\ell'} \neq v_\ell$. By the assumption of Case 2, we can choose such an $\ell' > 1$. If $\ell' < n$, swap the ids of $p_{\ell'}$ and p_n . This ensures that $v_n \neq v_\ell$.)

Then, define A to be $\{p_i : 1 \leq i \leq n - 1 \text{ and } v_i = v_\ell\}$ and B to be $\{p_i : 1 \leq i \leq n - 1 \text{ and } v_i \neq v_\ell\}$. It follows from the fact that not all of v_1, \dots, v_n are the same and the assumption of Case 2, that both teams are non-empty.

It follows from the definitions of A and B that, in either case, they form a partition of the processes p_1, \dots, p_{n-1} into two non-empty teams satisfying the following properties:

P1: $v_i \neq v_j$ for all $p_i \in A$ and $p_j \in B$, and

P2: $v_i \neq v_n$ for all $p_i \in A$.

We check that $Q_A(q_0, op_1, \dots, op_{n-1})$ and $Q_B(q_0, op_1, \dots, op_{n-1})$ satisfy the definition of $(n - 1)$ -recording.

To derive a contradiction, suppose there is a state $q \in Q_A \cap Q_B$. This means there is a sequence of distinct process ids i_1, \dots, i_α chosen from $\{1, \dots, n - 1\}$ with $p_{i_1} \in A$ and another sequence of distinct process ids j_1, \dots, j_β chosen from $\{1, \dots, n - 1\}$ with $p_{j_1} \in B$ such that the sequences $op_{i_1}, \dots, op_{i_\alpha}$ and $op_{j_1}, \dots, op_{j_\beta}$ both take object O from state q_0 to state q . Adding one more operation op_n to the end of these sequences would leave O in the same state q' . (See Figure 3(c).) By Lemma 15, $v_{i_1} = v_{j_1}$. This contradicts property P1. Thus, condition 1 of the definition of $(n - 1)$ -recording holds.

To derive a contradiction, suppose $q_0 \in Q_A$. Then, there is a sequence of distinct process ids i_1, \dots, i_α chosen from $\{1, \dots, n-1\}$ with $p_{i_1} \in A$ such that the sequence of operations $op_{i_1}, \dots, op_{i_\alpha}$ takes object O from state q_0 back to state q_0 . The two sequences of operations on O shown in Figure 3(d) both leave O in the same state. Thus, $v_{i_1} = v_n$, by Lemma 15, contradicting property P2. Thus, condition 2 of the definition of $(n-1)$ -recording is satisfied.

To derive a contradiction, suppose $q_0 \in Q_B$ and $|A| > 1$. Since $|A| > 1$, the teams must have been defined according to Case 2. Since $q_0 \in Q_B$, there is a sequence of distinct process ids j_1, \dots, j_β chosen from $\{1, \dots, n-1\}$ with $p_{j_1} \in B$ such that $op_{j_1}, \dots, op_{j_\beta}$ takes object O from state q_0 back to q_0 . So, in Case 2 of the definition of the teams, we chose $\ell = i_1$, where i_1, \dots, i_α is some sequence of distinct process ids chosen from $\{1, \dots, n\}$ such that $op_{i_1}, \dots, op_{i_\alpha}$ also takes object O from state q_0 back to q_0 . Since $i_1 = \ell \leq n-1$, we have $p_{i_1} \in A$. (We remark that this sequence's existence does not contradict the fact proved above that $q_0 \notin Q_A(q_0, op_1, \dots, op_{n-1})$, since this sequence may include the index n .)

Our goal is to show that $v_{i_1} = v_{j_1}$, which will contradict property P1. We use a case argument, showing that it is possible to apply Lemma 15 in each case. Let $I = \{k : 2 \leq k \leq n \text{ and } v_k = v_{i_1}\}$ and let $J = \{k : 2 \leq k \leq n \text{ and } v_k = v_{j_1}\}$. A step by a process whose index is in I or J extends the critical execution γ to a v_{i_1} - or v_{j_1} -valent execution, respectively. Moreover, a step by any process in I or J allows us to invoke Lemma 15 since the sets I and J do not include 1.

Case a: Suppose some $k \in J$ does not appear in i_1, \dots, i_α . Then, the two sequences of operations on O in Figure 3(e) leave O in the same state. Since $k \geq 2$, Lemma 15 implies that $v_{i_1} = v_k$. By definition of J , $v_k = v_{j_1}$. Thus, $v_{i_1} = v_{j_1}$.

Case b: Suppose there is some $k \in I$ that does not appear in j_1, \dots, j_β . By an argument symmetric to Case a, $v_{i_1} = v_{j_1}$.

Case c: Suppose $J \subseteq \{i_1, \dots, i_\alpha\}$ and $I \subseteq \{j_1, \dots, j_\beta\}$. We first argue that I is non-empty. If $i_1 > 1$, then $i_1 \in I$. Otherwise, $i_1 = 1$ and by the assumption of Case 2, there is some other process id k such that $v_k = v_{i_1}$ and this k is in I . A symmetric argument can be used to show that J is non-empty. Thus, both of the sequences i_1, \dots, i_α and j_1, \dots, j_β contain at least one of the ids in $\{2, \dots, n\}$. Since both sequences of operations shown in Figure 3(f) leave O in the same state q_0 , it follows from Lemma 15 that $v_{i_1} = v_{j_1}$.

In all three cases, $v_{i_1} = v_{j_1}$, contradicting Property P1. Thus, condition 3 of the definition of $(n-1)$ -recording holds. \square

In proving that T is $(n-1)$ -recording, we split $n-1$ of the processes into two teams according to the valency induced by their next step after the critical execution and assigned each process the operation they perform in this step. To show that these choices satisfy the definition of $(n-1)$ -recording, it was essential to have one process p_n "in reserve" that we could use to take one step in Figures 3(c) and 3(d). This step enables the crash of p_1 needed to prove Lemma 15, which shows that the two executions in those figures lead to the same outcome, thereby deriving the necessary contradiction. This is the reason we show that being $(n-1)$ -recording (rather than n -recording) is necessary for solving RC.

3.3 Relationship Between Consensus and Recoverable Consensus

Next, we prove a relationship between the characterizations of types that solve consensus and those that solve RC.

Theorem 16. *For $n \geq 4$, if a deterministic type T is n -discerning, then it is $(n-2)$ -recording.*

Proof. Let $q_0, A, B, op_1, \dots, op_n$ be chosen to satisfy the definition of n -discerning. Without loss of generality, assume that $\{p_1, \dots, p_{n-2}\}$ includes at least one process from each of A and B , and that $\{p_{n-1}, p_n\}$ includes at least one process from each team that contains more than one process. (The ids of the processes can be permuted to make this true.) We partition the processes $\{p_1, \dots, p_{n-2}\}$ into two non-empty teams $A' = A \cap \{p_1, \dots, p_{n-2}\}$ and $B' = B \cap \{p_1, \dots, p_{n-2}\}$.

We show that $Q_{A'}(q_0, op_1, \dots, op_{n-2})$ and $Q_{B'}(q_0, op_1, \dots, op_{n-2})$ satisfy the definition of $(n-2)$ -recording.

To derive a contradiction, assume $Q_{A'} \cap Q_{B'}$ contains some state q . Then, there are sequences i_1, \dots, i_α and j_1, \dots, j_β , each of distinct ids from $\{1, \dots, n-2\}$, such that $p_{i_1} \in A, p_{j_1} \in B$ and the sequences $op_{i_1}, \dots, op_{i_\alpha}$ and $op_{j_1}, \dots, op_{j_\beta}$ both take an object of type T from state q_0 to q . Operation op_n takes the object from state q to some state q' and returns some response r . By adding op_n to the end of each of the two sequences, we see the pair (r, q') is in both $R_{A,n}$ and $R_{B,n}$ in the definition of n -discerning, a contradiction. Thus, condition 1 of the definition of $(n-2)$ -recording is satisfied.

To derive a contradiction, assume $q_0 \in Q_{A'}$ and $|B'| > 1$. Since $|B| \geq |B'| > 1$, some process p_j is in $B \cap \{p_{n-1}, p_n\}$. Operation op_j takes an object of type T from q_0 to some state q and returns some response r . Thus, (r, q) is in the set $R_{B,j}$ of the definition of n -discerning. Since $q_0 \in Q_{A'}$, there is a sequence i_1, \dots, i_α of distinct ids chosen from $\{1, \dots, n-2\}$ such that $p_{i_1} \in A$ and the sequence $op_{i_1}, \dots, op_{i_\alpha}$ takes an object of type T from state q_0 back to the state q_0 . By adding op_j to the end of this sequence, we see that the pair (r, q) is also in $R_{A,j}$, contradicting the fact that $R_{A,j} \cap R_{B,j}$ must be empty, according to the definition of n -discerning. Thus, condition 2 of the definition of $(n-2)$ -recording is satisfied.

The proof of condition 3 is symmetric. □

Corollary 17. *A deterministic, readable object type T with consensus number at least n can solve recoverable consensus among $n-2$ processes. Thus, $cons(T) - 2 \leq rcons(T) \leq cons(T)$.*

The first inequality in the corollary is a consequence of Theorem 8 and 16. The second inequality follows from the fact that any algorithm that solves RC is also an algorithm that solves consensus.

For $n=3$, we can strengthen Theorem 16 and Corollary 17 as follows. See Appendix C the full version [13] for the proof.

Proposition 18. *If a deterministic, readable type is 3-discerning, then it is 2-recording. Thus, if $cons(T) = 3$ then $2 \leq rcons(T) \leq 3$.*

The following example shows that Theorem 16 cannot be strengthened when $n > 3$.

Proposition 19. *For all $n \geq 4$, there is a type that is n -discerning, but is not $(n-1)$ -recording.*

A complete proof is in Appendix D[13]. We sketch it here. We define a type T_n whose set of states is $\{(winner, row, col) : winner \in \{\mathbb{A}, \mathbb{B}\}, 0 \leq row < \lceil n/2 \rceil, 0 \leq col < \lfloor n/2 \rfloor\} \cup \{(\perp, 0, 0)\}$. T_n has two operations $op_{\mathbb{A}}$ and $op_{\mathbb{B}}$, and a read operation. Intuitively, if the object is initialized to $(\perp, 0, 0)$, *winner* keeps track of whether the first update was $op_{\mathbb{A}}$ or $op_{\mathbb{B}}$, while *col* and *row* store the number of times $op_{\mathbb{A}}$ and $op_{\mathbb{B}}$ have been applied. If $op_{\mathbb{A}}$ is performed more than $\lfloor n/2 \rfloor$ times or $op_{\mathbb{B}}$ is performed more than $\lceil n/2 \rceil$ times, the object “forgets” all the information it has stored by going back to state $(\perp, 0, 0)$. It is easy to verify that T_n is n -discerning but not $(n-1)$ -recording.

It follows easily from Proposition 19 combined with Theorems 3 and 14 that there are readable types whose RC numbers are strictly smaller than their consensus numbers.

Corollary 20. *For all $n \geq 4$, there is a deterministic, readable type T_n such that $rcons(T_n) < cons(T_n) = n$.*

On the other hand, there are also types whose RC numbers are equal to their consensus numbers. The next proposition also shows that every level of the RC hierarchy is populated, since there are types with consensus number n for all n .

Proposition 21. *For all n , there is a deterministic, readable type S_n such that $rcons(S_n) = cons(S_n) = n$.*

A complete proof is in Appendix E the full version [13]. We sketch it here. We define a type S_n whose set of possible states is $\{(winner, row) : winner \in \{\mathbb{A}, \mathbb{B}\}, 0 \leq row < n\}$. S_n has two operations $op_{\mathbb{A}}$ and $op_{\mathbb{B}}$, and a read operation. Intuitively, if the object is initialized to $(\mathbb{B}, 0)$, and then accessed by update operations, $winner$ records whether the first update was $op_{\mathbb{A}}$ or $op_{\mathbb{B}}$ and row counts the number of times $op_{\mathbb{B}}$ has been applied. If $op_{\mathbb{A}}$ is performed more than once or if $op_{\mathbb{B}}$ is performed more than $n - 1$ times, then the object “forgets” all the information it has stored by going back to state $(\mathbb{B}, 0)$. It is fairly straightforward to check that S_n is n -recording, but is not $(n + 1)$ -discerning. Thus, $n \leq rcons(S_n) \leq cons(S_n) \leq n$.

3.4 Recoverable Consensus Using Several Types

The (recoverable) consensus number of a set \mathcal{T} of object types is the maximum number of processes that can solve (recoverable) consensus using objects of those types, together with registers (or ∞ if there is no such maximum). A classic open question, originally formulated by Jayanti [27], is whether the standard consensus hierarchy is robust for deterministic types, i.e., whether $cons(\mathcal{T}) = \max\{cons(T) : T \in \mathcal{T}\}$. If this equation holds so, it is possible to study the power of a system equipped with multiple types by studying the power of each type individually. See [16, Section 9] for some history of the robustness question. Ruppert’s characterization (Theorem 3) was used to show the consensus hierarchy is robust for the class of deterministic, readable types. Similarly, our characterization allows us to show how the power of a set of deterministic, readable types to solve RC is related to the power of the individual types.

Theorem 22. *Let \mathcal{T} be a non-empty set of deterministic, readable types and suppose $n = \max\{rcons(T) : T \in \mathcal{T}\}$ exists. Then, $n \leq rcons(\mathcal{T}) \leq n + 1$. (If $\max\{rcons(T) : T \in \mathcal{T}\}$ does not exist, then $rcons(\mathcal{T}) = \infty$.)*

Proof. If $\max\{rcons(T) : T \in \mathcal{T}\}$ does not exist, then for any n there is an algorithm that solves RC using some type $T_n \in \mathcal{T}$. It follows that $rcons(\mathcal{T}) = \infty$. So for the remainder of the proof, assume the maximum does exist.

It follows from the definition that $rcons(\mathcal{T}) \geq rcons(T)$ for all $T \in \mathcal{T}$. Thus, $rcons(\mathcal{T}) \geq \max\{rcons(T) : T \in \mathcal{T}\}$.

We prove the other inequality by contradiction. Suppose $(n + 2)$ -process RC can be solved using types in \mathcal{T} . As in the proof of Theorem 14, there is a critical execution γ at the end of which each process is about to update the same object O of some type $T \in \mathcal{T}$. As in that proof, T is $(n + 1)$ -recording. By Theorem 8, there is an $(n + 1)$ -process RC algorithm using objects of type T and registers. So, $rcons(T) \geq n + 1 > n \geq rcons(\mathcal{T})$, a contradiction. \square

4 The Significance of Recoverable Consensus

Herlihy’s universal construction [25] builds a linearizable, wait-free implementation of *any* shared object using a consensus algorithm as a subroutine. It creates a linked list of all operations performed on the implemented object, and this list defines the linearization ordering. Berryhill, Golab and Tripunitara [6] observed that this result extends to the model with simultaneous crashes, simply by placing the linked list in non-volatile memory and using RC in place of consensus. Their model allows a part of shared memory to be volatile. Using that volatile

memory, their universal construction provides strictly linearizable implementations. (*Strict linearizability* [1] is similar to linearizability, with the requirement that an operation in progress when a process crashes is either linearized before the crash or not at all.) Without volatile shared memory, the history satisfies only the weaker property of *recoverable linearizability* (proposed in [6], with a correction to the definition in [29]).

Similarly, we observe that Herlihy’s universal construction also extends to the independent crash model. To execute an operation op , a process creates a node nd containing op (including its parameters). Then, it announces op by storing a pointer to nd in an announcement array. Other processes can then help add op to the list, ensuring wait-freedom. Processes use an instance of consensus to agree on the next pointer of each node in the list. A process executes a routine **Perform** that traverses the list. At each visited node, it proposes a value from the announcement array to the consensus algorithm for the node’s next pointer, until it discovers its own operation’s node nd has been appended. Processes choose which announced value to propose so that each process’s announced value is given priority in a round-robin fashion. This ensures each announced node is appended within a finite number of steps.

In our setting, all shared variables are non-volatile, and we use an algorithm for RC (such as the one in Section 3.1) in place of consensus. For simplicity, we use a standard assumption (as in, e.g., [3, 2, 15, 10, 32, 11, 18, 19]): when a process recovers from a crash, it executes a *recovery function*. This assumption is not restrictive; we could, alternatively, add the code of the recovery function at the beginning of the universal algorithm, thus forcing every process to execute this code before it actually starts executing a new operation. When a process p crashes and recovers, the recovery function checks if the last operation that p announced before crashing has been appended in the list and if not, it executes the code to append it. Specifically, the recovery function simply calls **Perform** for the last announced node of p . See Appendix F the full version [13] for pseudocode of the recoverable universal construction **RUniversal**.

As in Herlihy’s construction, the helping mechanism of **RUniversal** ensures wait-freedom. The recoverable implementations obtained using **RUniversal** satisfy *nesting-safe recoverable linearizability* (NRL) [3], which requires that a crashed operation is linearized within an interval that includes its crashes and recovery attempts. NRL implies *detectability* [3] which ensures that a process can discover upon recovery whether or not its last operation took effect, and guarantees that if it did, its response value was made persistent. Other well-known safety conditions for the crash-recovery setting include *durable linearizability* [26], which has been proposed for the system-crash failures model and requires that the effects of all operations that have completed before a crash are reflected in the object’s state upon recovery, and *persistent linearizability* [24], which has been proposed for a model where no recovery function is provided and requires that an operation interrupted by a crash can be linearized up until the invocation of the next operation by the same process. With minor adjustments these conditions are meaningful in our setting and **RUniversal** satisfies both of them.

Moreover, **RUniversal** has the following desirable property. Suppose an implementation I uses a linearizable object X in a system with halting failures, but no crash-recovery failures. We can transform I to an implementation I' by replacing every instance of X in I with an invocation of **RUniversal** (that implements X). Then, every trace produced by I' in a system with crash and recovery failures is also a trace of I using a linearizable object X in a system with halting failures. In this way, any algorithm designed for the standard asynchronous model with halting failures can be automatically transformed to another algorithm to run in the independent crash-recovery model, as long as we can solve RC.

The traditional consensus hierarchy gives us information about which implementations are possible (via universality), but also tells us some implementations are impossible. This is another reason to study the consensus hierarchy. Specifically, if $cons(T_1) < cons(T_2)$, then there is no wait-free implementation of object type T_2 from objects of type T_1 for more than $cons(T_1)$ processes [25]. We give an analogous result for the RC hierarchy. The proof is in

Appendix G. For the proof, see [13].

Theorem 23. *Let $n \leq rcons(T_2)$. If there is a wait-free, persistently linearizable implementation of T_2 from atomic objects of type T_1 (and registers) in a system of n processes with independent crashes, then $rcons(T_1) \geq n$.*

Corollary 24. *If $rcons(T_1) < rcons(T_2)$ then there is no wait-free, persistently linearizable implementation of T_2 from atomic objects of type T_1 and registers in a system of more than $rcons(T_1)$ processes with independent crashes.*

5 Discussion

In this paper, we studied solvability, without considering efficiency. A lot of research has focused on designing efficient recoverable transactional memory systems [34, 8, 7, 32, 9, 4]) and recoverable universal constructions [11, 15]. Wait-free solutions appear in [32, 11, 15]. Some [11, 15] are based on existing wait-free universal constructions [12, 14] for the standard shared-memory model with halting failures. All except [15], satisfy weaker consistency conditions than nesting-safe recoverable linearizability. Attiya *et al.* [3] gave a recoverable implementation of a Compare&Swap (CAS) object. Any concurrent algorithm from read/write and CAS objects can become recoverable by replacing its CAS objects with their recoverable implementation [3]. Capsules [5] can also be used to transform concurrent algorithms that use only read and CAS primitives to their recoverable versions. Many other general techniques [2, 19, 18] have been proposed for deriving recoverable lock-free data structures from their concurrent implementations.

Our work leaves open several questions. Is there a deterministic, readable type T with $rcons(T) = cons(T) - 2$? We saw in Corollary 17 that $cons(T) - rcons(T)$ can be at most 2 for deterministic, readable types. How big can this difference be for non-readable types?

It would be nice to close the gap between the necessary condition of being $(n - 1)$ -recording and the sufficient condition of being n -recording for the solvability of RC using deterministic, readable types. Perhaps a good starting point is to determine whether being 2-recording is actually necessary for solving 2-process RC. Finally, it would be interesting to characterize read-modify-write types capable of solving n -process RC (as was done in [33] for the standard consensus problem), and see whether the RC hierarchy is robust for deterministic, readable types (or for all deterministic types).

6 Acknowledgements

This research was conducted while Eric Ruppert was visiting the Université Paris Cité, with funding from IDEX-Université Paris Cité project BAD and ANR DUCAT project number -20-EC48-0006. Support was also provided by the Natural Sciences and Engineering Research Council of Canada, and HFRI under the 2nd Call for HFRI Research Projects to support faculty members and researchers (project number 3684).

References

- [1] Marcos K. Aguilera and Svend Frølund. Strict linearizability and the power of aborting. Technical Report HPL-2003-241, HP Laboratories Palo Alto, 2003. Available from <https://www.hpl.hp.com/techreports/2003/HPL-2003-241.html>.
- [2] Hagit Attiya, Ohad Ben-Baruch, Panagiota Fatourou, Danny Hendler, and Eleftherios Kosmas. Detectable recovery of lock-free data structures. In *Proc. 27th ACM Symposium on Principles and Practice of Parallel Programming*, pages 262–277, 2022.

- [3] Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. Nesting-safe recoverable linearizability: Modular constructions for non-volatile memory. In *Proc. ACM Symposium on Principles of Distributed Computing*, pages 7–16, 2018.
- [4] H. Alan Beadle, Wentao Cai, Haosen Wen, and Michael L. Scott. Nonblocking persistent software transactional memory. In *Proc. 25th ACM Symposium on Principles and Practice of Parallel Programming*, pages 429–430, 2020.
- [5] Naama Ben-David, Guy E. Blelloch, Michal Friedman, and Yuanhao Wei. Delay-free concurrency on faulty persistent memory. In *Proc. 31st ACM Symposium on Parallelism in Algorithms and Architectures*, pages 253–264, 2019.
- [6] Ryan Berryhill, Wojciech M. Golab, and Mahesh Tripunitara. Robust shared objects for non-volatile main memory. In *Proc. 19th International Conference on Principles of Distributed Systems*, volume 46 of *LIPICs*, pages 20:1–20:17, 2015.
- [7] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. *ACM SIGPLAN Notices*, 49(10):433–452, October 2014.
- [8] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. *ACM SIGARCH Computer Architecture News*, 39(1):105–118, March 2011.
- [9] Nachshon Cohen, Rachid Guerraoui, and Igor Zablotchi. The inherent cost of remembering consistently. In *Proc. 30th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 259–269, 2018.
- [10] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *Proc. 30th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 271–282, 2018.
- [11] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Persistent memory and the rise of universal constructions. In *Proc. 15th European Conference on Computer Systems*, pages 5:1–5:15, 2020.
- [12] Andreia Correia, Pedro Ramalhete, and Pascal Felber. A wait-free universal construction for large objects. In *Proc. 25th ACM Symposium on Principles and Practice of Parallel Programming*, pages 102–116, 2020.
- [13] Carole Delporte-Gallet, Panagiota Fatourou, Hugues Fauconnier, and Eric Ruppert. When is recoverable consensus harder than consensus? Full version available from <https://arxiv.org/abs/2205.14213>, 2022.
- [14] Panagiota Fatourou and Nikolaos D. Kallimanis. Highly-efficient wait-free synchronization. *Theory of Computing Systems*, 55(3):475–520, 2014.
- [15] Panagiota Fatourou, Nikolaos D. Kallimanis, and Eleftherios Kosmas. The performance power of software combining in persistence. In *Proc. 27th ACM Symposium on Principles and Practice of Parallel Programming*, pages 337–352, 2022.
- [16] Faith Fich and Eric Ruppert. Hundreds of impossibility results for distributed computing. *Distributed Computing*, 16(2–3):121–163, 2003.
- [17] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

- [18] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch, and Erez Petrank. NVTraverse: In NVRAM data structures, the destination is more important than the journey. In *Proc. 41st ACM Conference on Programming Language Design and Implementation*, pages 377–392, 2020.
- [19] Michal Friedman, Erez Petrank, and Pedro Ramalhete. Mirror: Making lock-free data structures persistent. In *Proc. 42nd ACM Conference on Programming Language Design and Implementation*, pages 1218–1232, 2021.
- [20] Wojciech Golab. The recoverable consensus hierarchy. In *Proc. 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 281–291, 2020. Extended version available from <https://arxiv.org/abs/1804.10597>.
- [21] Wojciech Golab and Danny Hendler. Recoverable mutual exclusion under system-wide failures. In *Proc. ACM Symposium on Principles of Distributed Computing*, pages 17–26, 2018.
- [22] Wojciech M. Golab and Danny Hendler. Recoverable mutual exclusion in sub-logarithmic time. In *Proc. ACM Symposium on Principles of Distributed Computing*, pages 211–220, 2017.
- [23] Wojciech M. Golab and Aditya Ramaraju. Recoverable mutual exclusion. In *Proc. ACM Symposium on Principles of Distributed Computing*, pages 65–74, 2016.
- [24] Rachid Guerraoui and Ron R. Levy. Robust emulations of shared memory in a crash-recovery model. In *Proc. 24th International Conference on Distributed Computing Systems*, pages 400–407, 2004.
- [25] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.
- [26] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *Proc. 30th International Symposium on Distributed Computing*, volume 9888 of *LNCS*, pages 313–327, 2016.
- [27] Prasad Jayanti. Robust wait-free hierarchies. *Journal of the ACM*, 44(4):592–614, July 1997.
- [28] Dénes König. Über eine Schlussweise aus dem Endlichen ins Unendliche. *Acta Litterarum ac Scientiarum Regiae Universitatis Hungaricae Franciscus-Josephinae: Sectio Scientiarum Mathematicarum*, 3:121–130, 1927. The required result also appears as Theorem 3 in chapter VI of König’s *Theory of Finite and Infinite Graphs*, Birkhäuser, Boston, 1990.
- [29] Nan Li. Detectable data structures for persistent memory. Master’s thesis, University of Waterloo, 2021. Available from <https://uwspace.uwaterloo.ca/handle/10012/16986>.
- [30] Wai-Kau Lo and Vassos Hadzilacos. All of us are smarter than any of us: Nondeterministic wait-free hierarchies are not robust. *SIAM Journal on Computing*, 30(3):689–728, 2000.
- [31] Gil Neiger. Failure detectors and the wait-free hierarchy. In *Proc. 14th ACM Symposium on Principles of Distributed Computing*, pages 100–109, 1995.
- [32] Pedro Ramalhete, Andreia Correia, Pascal Felber, and Nachshon Cohen. Onefile: A wait-free persistent transactional memory. In *Proc. 49th IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 151–163, 2019.

- [33] Eric Ruppert. Determining consensus numbers. *SIAM Journal on Computing*, 30(4):1156–1168, 2000.
- [34] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. *ACM SIGPLAN Notices*, 46(3):91–104, March 2011.

A Proof of Theorem 1

```

31 shared variables:
32   array  $Round[1..n]$  of registers, initially 0
33   array  $D[1..\infty]$  of registers, initially  $\perp$ 
34 DECIDE( $v$ )
35    $pref \leftarrow v$ 
36    $r \leftarrow 1$ 
37   loop
38     if  $Round[j] < r$  then
39        $Round[j] \leftarrow r$ 
40       if  $r > 1$  and  $D[r - 1] \neq \perp$  then
41          $pref \leftarrow D[r - 1]$ 
42       end if
43        $pref \leftarrow C_r.DECIDE(pref)$ 
44        $D[r] \leftarrow pref$ 
45       if  $\forall k, Round[k] \leq r$  then
46         return  $pref$ 
47       end if
48     else if  $r > 1$  and  $D[r - 1] \neq \perp$  then
49        $pref \leftarrow D[r - 1]$ 
50     end if
51      $r \leftarrow r + 1$ 
52   end loop
53 end DECIDE

```

Figure 4: Algorithm for process p_j to solve recoverable consensus with simultaneous crashes using instances of standard consensus C_1, C_2, \dots

We must show that RC is solvable for n processes with simultaneous crashes if and only if the standard consensus problem is solvable for n processes. The “only if” direction is trivial, since any algorithm for RC also solves consensus: just consider executions in which there are no crashes.

We prove the converse using the algorithm shown in Figure 4, which is similar to the algorithm in [20] for a bounded number of crashes. The RC algorithm uses multiple instances of the consensus algorithm, denoted C_1, C_2, \dots . Each process attempts to access C_1, C_2, \dots in turn, until it receives a result from one of them and sees that no process has yet moved on to the next object. The register $D[r]$ is used to record the output of C_r . An iteration of the loop with $r = i$ is called a *run of round i* . A process may run round i multiple times if it crashes and recovers. Process p_j records in register $Round[j]$ the largest r for which p_j has started to run round i . This variable is used to ensure that p_j does not access C_r a second time if it crashes during a run of round r .

Lemma 25. *The algorithm in Figure 4 satisfies the recoverable wait-freedom property.*

Proof. To derive a contradiction, suppose there is a time t after which there are no more crashes and processes continue to take steps without any process terminating. Let i be greater than the maximum entry in $Round[1..n]$ at time t . Any process that executes enough steps after t will either terminate, crash or reach round i . The processes that reach round i without crashing will satisfy the test at line 38. Among those processes, the first to complete line 45 will satisfy the test and terminate, a contradiction. \square

Observation 26. *For any j , the value in $Round[j]$ only increases.*

Proof. Only process p_j writes to $Round[j]$. Moreover, p_j writes a value r to $Round[j]$ (at line 39) only after seeing (at line 38) that the current value of $Round[j]$ is less than r . \square

Lemma 27. *For each i , no process invokes DECIDE on C_i more than once.*

Proof. Before p_j invokes DECIDE on C_i for the first time, p_j writes i to $Round[j]$. By Observation 26, $Round[j] \geq i$ at all times after that invocation. Thus, any subsequent test by p_j at line 38 of a run of round i will fail, and p_j will never invoke DECIDE on C_i again. \square

Since C_i is accessed correctly, even if processes crash and recover, it follows that calls to C_i .DECIDE cannot return different values and that the common output value is one of the input values to one of the calls to C_i .DECIDE. In particular, this means that all values written to $D[i]$ at line 44 are identical.

Lemma 28. *Let P_0 be the set of input values to the RC algorithm. For $r \geq 1$, let P_r be the set of all values that are either written into $D[r]$, or stored in the local variable $pref$ of some process when it completes a run of round r by terminating at line 46 or reaching the end of the iteration of the loop. For $r \geq 1$, $P_r \subseteq P_{r-1}$.*

Proof. Let P'_r be the set of all values that are in the local variable $pref$ of some process at some time during a run of round r by that process. Since $P_r \subseteq P'_r$, we must simply show that $P'_r \subseteq P_{r-1}$. Let $v \in P'_r$. Setting $pref$ at line 43 does not add any new value to P'_r , by the validity property of the consensus algorithm C_r . Thus, some process's $pref$ was either equal to v at the beginning of a run of round r , or set to v at line 41 or 49. We show that in each of these cases, $v \in P_{r-1}$. If $r = 1$ and $pref$ is v at the start of a run of round 1, then $pref$ was set to v at line 35, so $v \in P_0$. If $r > 1$ and $pref$ is v at the start of a run of round r , then $pref$ was v at the end of p_j 's run of round $r - 1$, so $v \in P_{r-1}$. If $pref$ is set to v at line 41 or 49, then $r > 1$ and v was stored in $D[r - 1]$, so $v \in P_{r-1}$. \square

The validity property follows from Lemma 28: if a process returns x at line 46 in a run of round r , then $x \in P_r \subseteq P_0$. Thus, all output values are in the set P_0 of input values.

Lemma 29. *The algorithm in Figure 4 satisfies the agreement property.*

Proof. Consider an execution in which some processes return a value. Let i be the minimum number such that some process returns a value at line 46 of a run of round i . Any process that returns a value at line 46 during a run of round i returns the value produced by C_i at line 43. By Lemma 27 and the agreement property of C_i , all such processes return the same value x , and x is the only non- \perp value that can ever be stored in $D[i]$.

The process that returns a value x at line 46 in its run of round i first writes x into $D[i]$ at line 44 and then reads values less than or equal to i from each entry of $Round[1..n]$ at line 45. By Observation 26, no entry of $Round$ is greater than i at any time before x is first written into $D[i]$.

We show that P_{i+1} (as defined in Lemma 28) can only contain x . Consider a process p_j that either writes to $D[i + 1]$ or completes a run of round $i + 1$. We consider two cases. First, suppose $Round[j] \geq r$ at line 38 of p_j 's run of round $i + 1$. By the argument of the previous paragraph, x has already been written into $D[i]$, so p_j updates its $pref$ to x at line 49. Otherwise, $Round[j] < r$ at line 38 of a run of round $i + 1$. In this case, line 39 must be performed after x has been written to $D[i]$, as argued in the previous paragraph. Thus, p_j updates its $pref$ to x at line 41.

Since all processes p_j that execute round $i + 1$ update $pref$ to x , it follows that all inputs to C_{i+1} are x . By the validity property of C_{i+1} , processes can only update their $pref$ to x at line 43 of a run of round $i + 1$, and can only write the value x into $D[i + 1]$.

Thus, $P_{i+1} \subseteq \{x\}$. Consider any round $i' \geq i + 1$. By Lemma 28, $P_{i'} \subseteq P_{i+1} \subseteq \{x\}$. In particular, this means that any value that is returned in any round $i' \geq i + 1$ must also be x . \square

This completes the proof of Theorem 1. The algorithm in Figure 4 uses an unbounded number of instances of consensus. In the full version of [20], Golab showed that this is indeed necessary for such a construction.

B Using Recoverable Team Consensus to Solve Recoverable Consensus

Suppose a collection of processes is partitioned into two non-empty teams. The recoverable team consensus problem is the same as the RC problem, except with the precondition that all input values for processes on the same team must be the same. The following proposition, used in the proof of Theorem 8, can be proved in the same way as the analogous claim in [31, 33] for standard consensus, but we include the proof here for the sake of completeness.

Proposition 30. *If objects of type T and registers can be used to solve recoverable team consensus among n processes, then objects of type T and registers can be used to solve recoverable consensus among n processes.*

Proof. Assume we have a recoverable team consensus for n processes divided into two non-empty teams A and B . We use induction on k show that k processes can solve RC using objects of type T and registers.

For $k = 1$, this is trivial: a process can simply return its own input value.

Let $1 < k \leq n$. Assume the claim holds for fewer than k processes. We construct an algorithm to solve RC for k processes. Split the k processes into two non-empty teams A' and B' such that $|A'| \leq |A|$ and $|B'| \leq |B|$. We use two RC algorithms $R_{A'}$ and $R_{B'}$ for $|A'|$ and $|B'|$ processes, respectively. These algorithms can be built from objects of type T and registers, by the induction hypothesis since $|A'|$ and $|B'|$ are less than k . Each process first runs the RC algorithm for its team, and uses the output from it as the input to a recoverable team consensus algorithm TC to produce the final output. (Note that the n -process recoverable team consensus algorithm still works if only k processes use it; we think of the other $n - k$ processes simply taking no steps.)

The agreement property of $R_{A'}$ and $R_{B'}$ ensure that the precondition of the recoverable team consensus algorithm is satisfied. The agreement property of the k -process RC algorithm follows from the agreement property of TC . The recoverable wait-freedom and validity properties follow from the corresponding properties of $R_{A'}$, $R_{B'}$ and TC . \square

C Proof of Proposition 18

Proof. Let $A, B, q_0, op_1, op_2, op_3$ be chosen to satisfy the definition of 3-discerning for the type T . Without loss of generality, assume $A = \{p_1\}$ and $B = \{p_2, p_3\}$. Let $A' = \{p_1\}$ and $B' = \{p_2\}$. We show that A', B', q_0, op_1, op_2 satisfy the definition of 2-recording. Since $|A'| = |B'| = 1$, conditions 2 and 3 of the definition of 2-recording are trivially satisfied. The argument that condition 1 is satisfied is similar to the proof of Theorem 16: if some state q can be reached by two sequences starting with operations on opposite teams, we can append op_3 to the end of each to show that $R_{A,3} \cap R_{B,3} \neq \emptyset$, contradicting the definition of 3-discerning.

Since T is 2-recording, Theorem 8 implies that $2 \leq rcons(T)$. Moreover $rcons(T) \leq cons(T) = 3$, since any algorithm for RC also solves consensus. \square

D Proof of Proposition 19

We use the following definitions of Herlihy [25]. Operations op_i and op_j commute from state q_0 if the sequences op_i, op_j and op_j, op_i take the object from q_0 to the same state q . Operation


```

54   $op_A$ 
55    if  $winner = \perp$  then
56       $winner \leftarrow A$ 
57      return  $A$ 
58    else
59       $result \leftarrow winner$ 
60       $col \leftarrow (col + 1) \bmod \lfloor n/2 \rfloor$ 
61      if  $col = 0$  then
62         $winner \leftarrow \perp$ 
63         $row = 0$ 
64      end if
65      return  $result$ 
66    end if
67  end  $op_A$ 

68   $op_B$ 
69    if  $winner = \perp$  then
70       $winner \leftarrow B$ 
71      return  $B$ 
72    else
73       $result \leftarrow winner$ 
74       $row \leftarrow (row + 1) \bmod \lfloor n/2 \rfloor$ 
75      if  $row = 0$  then
76         $winner \leftarrow \perp$ 
77         $col = 0$ 
78      end if
79      return  $result$ 
80    end if
81  end  $op_B$ 

```

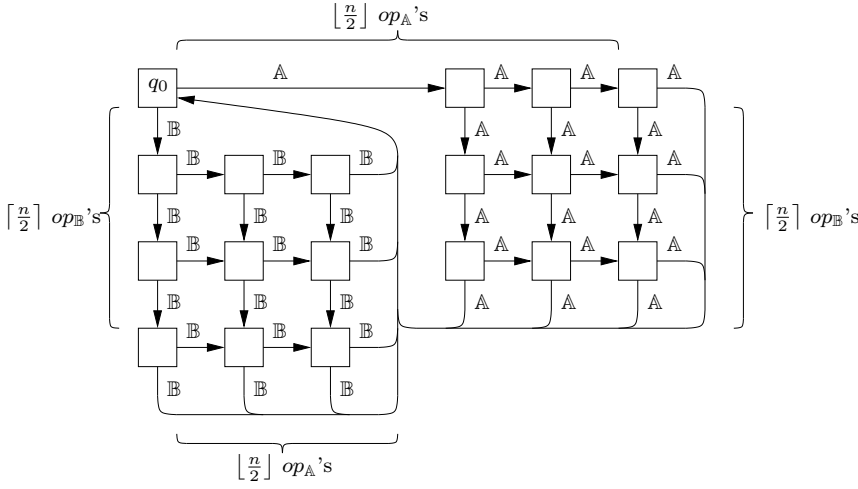


Figure 5: Behaviour of type T_n used in the proof of Proposition 19. A transition diagram for the object when $n = 6$ is shown, where the op_A and op_B operations are shown by the horizontal and vertical arrows leaving each state, respectively. Each transition is labelled by the output value of the operation that causes the transition.

op_i overwrites op_j from q_0 if the sequences op_i, op_j, op_i take the object from q_0 to the same state q . It is easy to check that if q_0, op_1, op_2 satisfy the definition of 2-recording, then op_1 and op_2 cannot commute from q_0 nor can one overwrite the other from q_0 .

Proof. We define the type T_n as follows. The set of possible states is $\{(winner, row, col) : winner \in \{A, B\}, 0 \leq row < \lfloor n/2 \rfloor, 0 \leq col < \lfloor n/2 \rfloor\} \cup \{(\perp, 0, 0)\}$. T_n supports two operations op_A and op_B , as well as a read operation. If an update operation is applied to an object in state $(winner, row, col)$, it executes the code in Figure 5 atomically to update the state and return a result.

To see that T_n is n -discerning, let $q_0 = (\perp, 0, 0)$ and partition processes into team A of size $\lfloor n/2 \rfloor$ and team B of size $\lceil n/2 \rceil$. Assign op_A to all processes on team A and op_B to all processes on team B . Then, if any sequence of operations assigned to distinct processes is applied to an object starting in state q_0 , every operation's result will be the name of the team of the process that performed the first operation in the sequence. This is because the state of the object can return to q_0 after a process on one team takes the first step only after *all* processes on the other team have taken a step. Thus, for any j , all pairs in $R_{A,j}$ will be of the form $(A, *)$ and all pairs in $R_{B,j}$ will be of the form $(B, *)$, so $R_{A,j} \cap R_{B,j} = \emptyset$.

It remains to show that T_n is not $(n-1)$ -recording. To derive a contradiction, suppose there is some $q_0, A, B, op_1, \dots, op_{n-1}$ that satisfy the definition of $(n-1)$ -recording.

If $q_0 \neq (\perp, 0, 0)$, then any pair of operations either commute or overwrite, so even the definition of 2-recording is not satisfied. So q_0 must be $(\perp, 0, 0)$. If two processes p_i and p_j on opposite teams are assigned the same operation, then op_i and op_j would both take an object from state q_0 to the same state, violating condition 1 of the definition of $(n - 1)$ -recording. Thus, without loss of generality, all processes on team A are assigned $op_{\mathbb{A}}$ and all processes on team B are assigned $op_{\mathbb{B}}$.

If $|A| \geq \lfloor n/2 \rfloor$, then allowing one process on team B to take a step followed by $\lfloor n/2 \rfloor$ processes on team A would take the object from state q_0 back to q_0 , so $q_0 \in Q_B$. This violates condition 3 of the definition of $(n - 1)$ -recording since $|A| \geq \lfloor 4/2 \rfloor = 2$. Thus, $|A| \leq \lfloor n/2 \rfloor - 1$.

Similarly, if $|B| \geq \lceil n/2 \rceil$, then allowing one process on team A to take a step followed by $\lceil n/2 \rceil$ processes on team B would take the object from state q_0 back to q_0 , so $q_0 \in Q_A$. This violates condition 2 of the definition of $(n - 1)$ -recording, since $|B| \geq \lceil 4/2 \rceil = 2$. Thus, $|B| \leq \lceil n/2 \rceil - 1$.

Hence, $n - 1 = |A| + |B| \leq (\lfloor n/2 \rfloor - 1) + (\lceil n/2 \rceil - 1) = n - 2$, a contradiction. \square

E Proof of Proposition 21

Proof. For $n = 1$, let S_1 be a type that provides only a read operation.

For $n \geq 2$, we define type S_n as follows. The set of possible states is $\{(winner, row) : winner \in \{\mathbb{A}, \mathbb{B}\}, 0 \leq row < n\}$. S_n supports two operations $op_{\mathbb{A}}$ and $op_{\mathbb{B}}$, as well as a read operation. If an update operation is applied to an object in state $(winner, row)$, it executes the code in Figure 6 atomically to update the state and return a result.

We first argue that S_n is n -recording. Let $q_0 = (\mathbb{B}, 0)$, $A = \{p_1\}$, $B = \{p_2, \dots, p_n\}$, $op_1 = op_{\mathbb{A}}$, $op_2 = op_3 = \dots = op_n = op_{\mathbb{B}}$. Then, $Q_A = \{(\mathbb{A}, row) : 0 \leq row < n\}$ and $Q_B = \{(\mathbb{B}, row) : 0 \leq row < n\}$ satisfy all the conditions of the definition of n -recording. It follows from Theorem 8 that $rcons(S_n) \geq n$.

Next, we argue that S_n is not $(n + 1)$ -discerning, in order to prove that $cons(S_n) \leq n$. To derive a contradiction, suppose S_n is $(n + 1)$ -discerning. Operations assigned to processes on opposite teams cannot commute or overwrite if they are performed on an object initially in state q_0 . (If two operations op_i and op_j assigned to opposite teams commuted, then for the two sequences op_i, op_j and op_j, op_i , the operation op_i gets the same result ack in both sequences, and both sequences leave the object in the same state. This would violate the definition of $(n + 1)$ -discerning. A similar argument applies for overwriting operations.) It is easy to check that q_0 must therefore be $(\mathbb{B}, 0)$ and processes on one team (without loss of generality, team A) must be assigned the operation $op_{\mathbb{A}}$ and processes on the other team B must be assigned the operation $op_{\mathbb{B}}$. If $|A| \geq 2$, then the sequences $op_{\mathbb{A}}, op_{\mathbb{A}}, op_{\mathbb{B}}$ and $op_{\mathbb{B}}$ performed on an object would both take an object from state $(\mathbb{B}, 0)$ to state $(\mathbb{B}, 1)$ and would return the same result for $op_{\mathbb{B}}$, violating the definition of $(n + 1)$ discerning. Thus, there must be just one process on team A . So, $|B| = n$. Consider the following two sequences of operations.

- All processes on team B perform $op_{\mathbb{B}}$ followed by the process on team A performing $op_{\mathbb{A}}$.
- The process on team A performs $op_{\mathbb{A}}$.

Both sequences take the object from state q_0 to $(\mathbb{A}, 0)$ and return the same result to $op_{\mathbb{A}}$, violating the definition of $(n + 1)$ -discerning. This contradiction completes the proof. \square

F Universal Construction

The pseudocode for our slightly modified version of Herlihy's universal construction is given in Figure 7. Each list node contains the following fields.

```

82  $op_A$ 
83   if  $(winner, row) = (B, 0)$  then
84      $winner \leftarrow A$ 
85   else
86      $winner \leftarrow B$ 
87      $row \leftarrow 0$ 
88   end if
89   return ack
90 end  $op_A$ 

91  $op_B$ 
92    $row \leftarrow (row + 1) \bmod n$ 
93   if  $row = 0$  then
94      $winner \leftarrow B$ 
95   end if
96   return ack
97 end  $op_B$ 

```

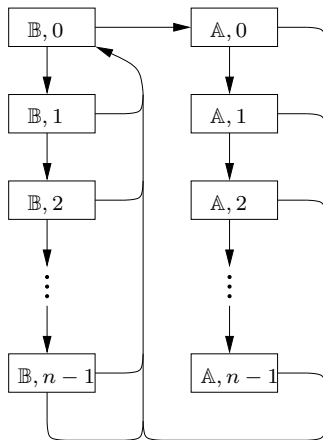


Figure 6: Behaviour of type S_n used in proof of Proposition 21. A transition diagram is shown, where the op_A and op_B operations are shown by the horizontal and vertical arrows leaving each state, respectively. All operations return ack.

```

98 shared variables:
99   Announce[1..n] of registers, each entry initially points to the dummy node at the beginning of the list
100  Head[1..n], each entry initially points to a dummy node at the beginning of the list

101 APPLYOPERATION // ensures that node Announce[i] is added to the list and returns result of that node's operation
102   while Announce[i]->seq = 0 // keep trying until my operation has been added to the list
103     priority ← (Head[i]->seq + 1) mod n // id of process who has priority for next list position
104     if Announce[priority]->seq = 0 then // check if process with id priority needs help
105       pointer ← Announce[priority] // try to add operation of process with id priority
106     else
107       pointer ← Announce[i] // try to add my own operation
108     end if
109     winner ← DECIDE(Head[i]->next, pointer) // propose pointer to RC instance associated with next pointer of node
110     // fill in the information fields of winner, the next node in the list.
111     ⟨winner->newState, winner->response⟩ ← APPLY(winner->op, Head[i]->newState)
112     winner->seq ← Head[i]->seq + 1
113     Head[i] ← winner // advance to next node
114   end while
115   return Announce[i]->response
116 end APPLYOPERATION

117 UNIVERSAL(op) // perform op on implemented object and return result
118   nd ← pointer to new list node
119   nd->op ← op // op includes name of operation to apply and its arguments
120   nd->seq ← 0
121   Announce[i] ← nd
122   for j ← 0..n - 1 // make sure Head[i] pointer is not too out of date
123     if Head[j]->seq > Head[i]->seq then
124       Head[i] ← Head[j]
125     end if
126   end for
127   return APPLYOPERATION
128 end UNIVERSAL

129 RECOVER
130   return APPLYOPERATION
131 end RECOVER

```

Figure 7: Universal construction pseudocode for process p_i .

- *seq* is initially 0 but is changed to the node's position in the list once the node is added to the list.
- *op* is the operation on the implemented object represented by the node; this includes the name of the operation and any arguments to it.
- *newState* is the state of the implemented object after the operations on the list up to and including this node have been applied to it.
- *response* is the result of the operation represented by the node.
- *next* is an instance of RC that will be used to agree upon the next node in the list.

Initially, the list contains a single dummy node whose sequence number is 1 and whose *newState* field stores the initial state of the implemented object.

We remark that a process that crashes and recovers might access the RC instance associated with the *next* pointer of a node multiple times with different input values. So, we should use the

mechanism described in the introduction to mask this behaviour and ensure that the process's inputs to the RC instance are identical.

G Proof of Theorem 23

Proof. Let A be an n -process algorithm for RC that uses atomic objects of type T_2 and registers. Construct an algorithm A' by replacing each object of type T_2 by a persistent linearizable implementation from atomic objects of type T_1 and registers.

We first argue that A' satisfies the recoverable wait-freedom property. If a process continues to take steps without crashing, it will eventually complete each operation it calls on a simulated object of type T_2 , since the implementation of T_2 is wait-free. Thus, it will eventually produce an output, since A satisfies recoverable wait-freedom.

It remains to show that A' satisfies the agreement and validity properties of RC. Let α' be any execution of A' . We construct a corresponding execution α of A as follows. Remove all internal steps of the implementation of T_2 (i.e., all steps of a process between an invocation step on a T_2 object and its subsequent response or process crash, or to the end of the execution if there is no such response or crash). Each simulated operation on an object of type T_2 in α' that is not linearized must not have a response in α' . We also remove its invocation when forming α . For each remaining operation on a T_2 object that has a response in α' , we “contract” the operation so that its invocation and response occur immediately after each other at the linearization point of the operation. Finally, we consider operations on T_2 objects that are linearized but have no response in α' , either because the process executing the operation crashes or does not take enough steps to complete the simulated operation. We move the invocation step to the linearization point and add a response step immediately afterwards. If the linearization point is *after* the crash that occurred while the operation was pending, then we also shift this crash step immediately after the response step.

It is easy to check that the constructed execution α is a legal execution of A with atomic objects. In particular, the sequence of steps taken by any process is the same in α as it is in α' (except for the removal of some invocations of operations on T_2 objects that do not terminate, either because they occur immediately before a crash of the process or because the process ceases taking steps). Thus, it will satisfy agreement and validity. The execution α contains the same output steps as α' , so α' also satisfies agreement and validity. \square

H $rcons(stack) = 1$

We use a valency argument to show that $rcons(stack) = 1$, i.e., that two processes cannot solve RC using stacks and registers. To derive a contradiction, assume there is an algorithm A for two processes to solve RC using stacks and registers. As in the proof of Theorem 14, we define valency with respect to a set \mathcal{E}_A of executions of A in which p_2 never crashes and, in any prefix of an execution, the number of crashes by p_1 is less than or equal to the number of steps taken by p_2 . By the same argument as in Theorem 14, there is a critical execution γ , and the next step specified by the algorithm for every process p_i is an operation op_i on a single stack O . Let v_i be the valency of the execution that is obtained by allowing p_i to perform its next step op_i after γ . As argued in Theorem 14, v_1 must be different from v_2 . The remainder of the proof is a case analysis, similar to Herlihy's proof that $cons(stack) = 2$ [25]. See Figure 8, in which the sequence of elements on the stack are shown from bottom to top in the order they are pushed, and α represents a (possibly empty) sequence of elements.

If both op_1 and op_2 are both POP, then the steps commute (Figure 8(a)). If op_1 is a PUSH(v) and op_2 is a POP, and the stack is empty at the end of γ , then op_1 overwrites op_2 (Figure 8(b)). In either of these cases, we have $v_1 = v_2$ by Lemma 15 (which applies even when O is not readable). This is the desired contradiction.

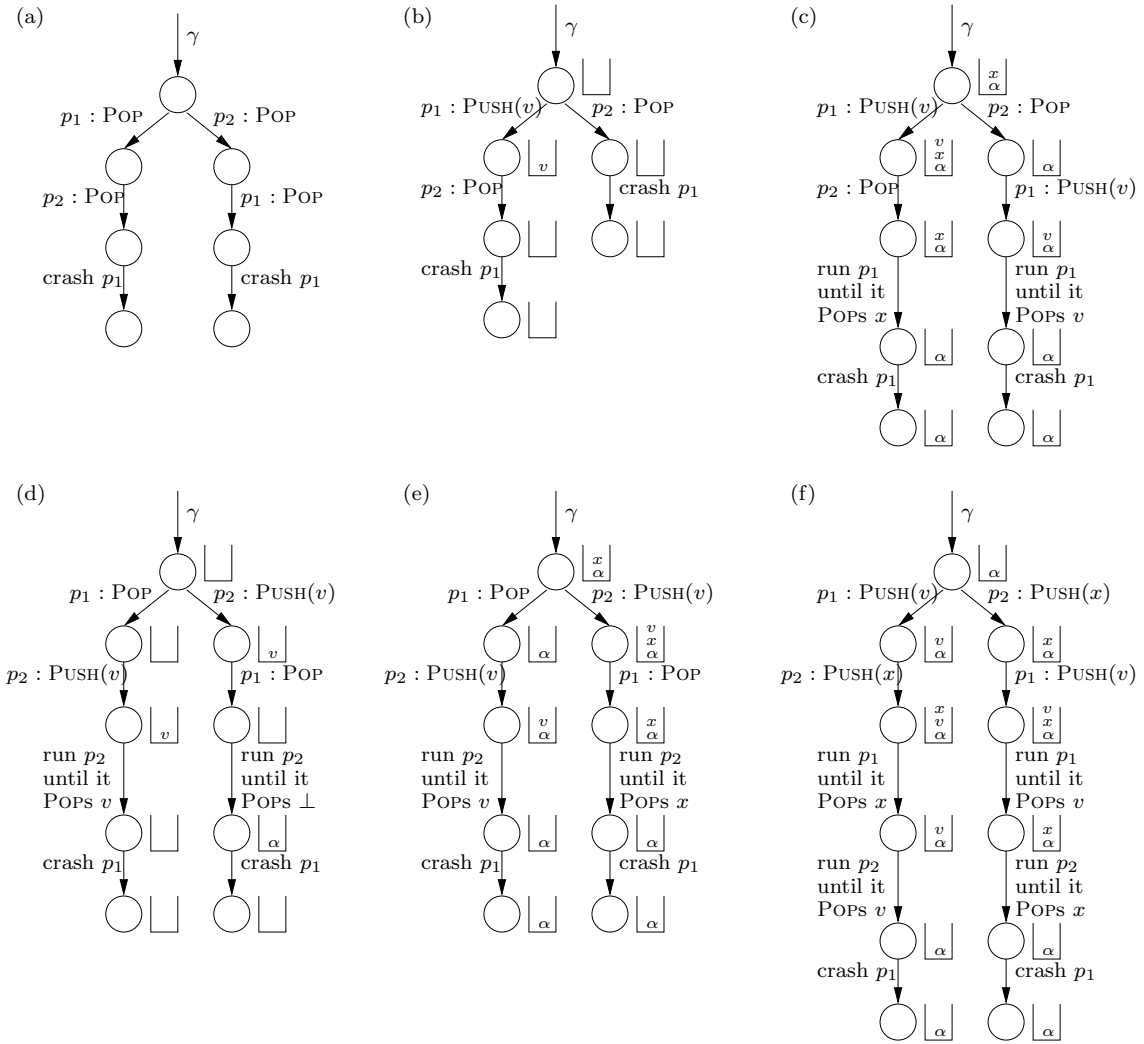


Figure 8: Impossibility of 2-process recoverable consensus using a stack.

If op_1 is a $PUSH(v)$ and op_2 is a POP and the stack is non-empty at the end of γ , then consider the two extensions of γ shown in Figure 8(c). After the operations op_1 and op_2 are done (in opposite order in the two extensions), the only differences between the two resulting states of the system are the local state of p_2 and the element that is on the top of the stack. Thus, if p_1 continues to run, it must run until it pops that top element; otherwise it would output the same value in both extensions, contradicting the fact that one is v_1 -valent and the other is v_2 -valent. During this solo execution by p_1 it takes the same steps in both extensions. Since p_2 has taken a step in both extensions, we can then crash p_1 . After p_1 crashes, the states of the system in the two extensions are identical except for p_2 's state. Thus, if p_1 recovers and executes A to completion, it must output the same value in the two extensions, contradicting the fact that the two extensions have different valencies.

The other cases shown in Figure 8(d) to 8(f) are argued similarly to the preceding case, and this completes the proof.

A similar argument could be used to show that $rcons(queue) = 1$.