On the power of shared object types to implement one-resilient Consensus

Wai-Kau Lo*
wklo@cs.utoronto.ca

Vassos Hadzilacos†
vassos@cs.utoronto.ca

Abstract

In this paper we study the ability of shared object types to implement Consensus in asynchronous shared-memory systems where at most one process may crash. More specifically, we consider the following question: Let $n \geq 3$ and $\mathcal{S}$ be a set of object types that can be used to solve one-resilient Consensus among n processes. Can $\mathcal{S}$ always be used to solve one-resilient Consensus among $n-1$ processes? We prove that for $n = 3$ the answer is negative, even if $\mathcal{S}$ consists only of *deterministic* types. (This strengthens an earlier result by the first author proving the same fact for *nondeterministic* types.) We also prove that, in contrast, for $n > 3$ the answer to the above question is affirmative.

1 Background and overview

In this paper we consider some questions concerning fault-tolerant implementations of Consensus in asynchronous shared-memory systems. In such systems, some number of processes communicate with each other by accessing shared typed objects. Processes take steps in a completely asynchronous manner. In one step, a process may invoke an operation on a shared object. This causes the object to atomically change its state and return a response to the process invoking the operation. The new state entered by the object and the response returned to the operation are determined by the specification of the type to which the object belongs. A process may *crash* — i.e., stop taking steps before completing its execution. A process that does not crash is called *correct*.

In this model of computation, an algorithm involving n processes is *t-resilient*, for $t < n$, if each correct process can complete any task it initiates, as long as no more than t processes crash. The algorithm is *wait-free* if it is $(n-1)$-resilient. Thus, in a wait-free algorithm, any correct process can complete its task after a finite number of its own steps regardless of the progress made by other processes. In fact, it must do so, even if all other processes crash.

In the Consensus problem, each process has a private input value and eventually may decide (irrevocably) on some value. An algorithm $\mathbf{A}$ for n processes *implements* (or *solves*) t-resilient Consensus among $n > t$ processes if every execution of $\mathbf{A}$ satisfies the following properties: (i) no two processes decide different values (Agreement); (ii) the value decided by any process is the input value of some process (Validity); and (iii) if no more than t processes crash in the execution then, eventually, each correct process must decide (Termination). Consensus is a fundamental problem in fault-tolerant distributed computing. It captures the essence of many practical problems that require some form of agreement. Thus, it is important to understand when this problem is solvable.

A set of types $\mathcal{S}$ *implements* (or *solves*) t-resilient Consensus among n processes, if there is an algorithm $\mathbf{A}$ that implements t-resilient Consensus among n processes and every object used by $\mathbf{A}$ belongs to a type in $\mathcal{S}$. Throughout this paper we assume that $\mathcal{S}$ contains type register; objects of type register are shared registers that support read and write operations (henceforth called simply "registers").

Let $\mathcal{S}$ be a set of types that implements t-resilient Consensus among n processes, for $n-1 > t$. It is easy to see that $\mathcal{S}$ can be used to solve t-resilient Consensus among $n+1$ processes (and thus any number of processes greater than n). It is not clear whether $\mathcal{S}$ can also be used to solve t-resilient Consensus among

*Supported by a Canadian Commonwealth Scholarship.

†Supported by a grant from the Natural Sciences and Engineering Research Council of Canada and a fellowship from the Engineering and Physical Sciences Research Council of the U.K.

Authors' address: Department of Computer Science, University of Toronto, 6 King's College Rd., Toronto, Ontario, Canada, M5S 3G4.

$n-1$ processes (and thus any number of processes in the range $t+1$ and $n-1$). Now, reaching Consensus appears more difficult, since the algorithm can rely on fewer correct processes ($n-t-1$ instead of $n-t$). Thus, *prima facie* it appears possible that a set of types is strong enough to solve t-resilient Consensus among n processes, but too weak to solve t-resilient Consensus among $n-1$ processes.

The question of whether this is possible was studied by Chandra *et al.* [3]. They showed that, for any integers n and t such that $n-1 > t \geq 2$, any set of types strong enough to solve t-resilient Consensus among n processes, is also strong enough to solve t-resilient Consensus among $n-1$ processes. The case of $t=1$ was left open in that investigation. This special case is an important one since, in practice, handling a single failure is often an adequate degree of fault-tolerance [13]. In a subsequent paper, the first author considered this special case and proved that the restriction to $t \geq 2$ is necessary for the above-cited result of [3]. Specifically, in [10] he exhibited a *nondeterministic* type wor with the following property: Using only wor objects and registers, it is possible to implement one-resilient Consensus for three, but not for two, processes. In this paper we complete the study of the special case of $t=1$, focusing on the ability of object types to solve one-resilient Consensus.

The rest of the paper is organised as follows: In Section 2, we strengthen the result of [10] by exhibiting a *deterministic* type that can be used (together with registers) to implement one-resilient Consensus for three processes, but not for two processes. This object type has another interesting property: To our knowledge it is the only known *deterministic* type at level one of the Consensus hierarchy [5] which cannot be implemented in a one-resilient manner for three or more processes using only registers.[1] In Section 3, on the other hand, we show that for any $n \geq 4$, a set of (deterministic or nondeterministic) types strong enough to implement one-resilient Consensus among n processes is also strong enough to implement one-resilient Consensus among $n-1$ processes. We conclude in Section 4.

The model of shared objects used in this paper is a standard one [5], and we do not detail it in this extended abstract. Due to space restrictions, all proofs have been omitted. Full details may be found in a technical report that is available electronically [11].

[1]The level of a type in the Consensus hierarchy is the maximum positive integer n such that we can implement wait-free (i.e., $(n-1)$-resilient) Consensus among n processes using only objects of that type and registers. (If there is no such maximum, the level of the type in the Consensus hierarchy is ∞.) The significance of this hierarchy lies in the following fundamental result, due to Herlihy [5]: If a type T is at level n of the Consensus hierarchy, then it is possible to use only objects of type T and registers to give a wait-free implementation of *any* object type in a system of n processes.

2 The (exceptional) case of three processes

In this section we introduce a deterministic type, named dor[2], that (together with registers) is strong enough to solve one-resilient Consensus for three processes, but not for two processes. The type is defined in Section 2.1. We prove that it is strong enough to solve one-resilient Consensus for three processes in Section 2.2. Finally, we prove that it is too weak to solve one-resilient Consensus for two processes (equivalently: wait-free Consensus for two processes) in Section 2.3.

2.1 Specification of dor

WK-2: Boldface dor

The specification of dor is motivated by a specific algorithm that we have in mind for implementing one-resilient Consensus among three processes using an object of this type and registers. Thus, it is best to explain dor alongside an informal description of that algorithm.

A process can apply two kinds of operations to a dor object: "enroll" an integer in the set $\{0, \ldots, 5\}$, and "query" whether 0 was enrolled before 1. In the algorithm that implements one-resilient Consensus among three processes Q_0, Q_1 and Q_2, each process enrolls two integers to a dor object O: Q_0 enrolls 0 and 3, Q_1 enrolls 1 and 4, and Q_2 enrolls 2 and 5. The three processes observe a particular protocol in enrolling their integers. Q_0 and Q_1 observe the same protocol, while Q_2 follows a slightly different one. Specifically, each of Q_0 and Q_1 enrolls its first integer in O and waits until at least one of the other two processes has also enrolled its first integer. It then enrolls its second integer and, again, waits until one of the other two processes has also done so. In contrast, Q_2 *first* waits for one of Q_0 and Q_1 to enroll its first integer in O and then it enrolls its own first integer. Similarly, Q_2 waits for one of the other two processes to enroll its second integer and then enrolls its own second integer. Thus, Q_0 and Q_1 follow the pattern: enroll, wait, enroll, wait; while Q_2 follows the pattern: wait, enroll, wait, enroll. Note that, as a result of this protocol, the first integer to be enrolled is either 0 or 1.

After enrolling its integers in this manner, each process Q_k "queries" O to determine whether 0 was enrolled before 1. This allows the three processes to agree on which of the two integers was enrolled first — and thus to agree on the identity of one of Q_0 and Q_1, so that if they agree on Q_k, then Q_k has taken at least one step (the step required to enroll k). By requiring each of Q_0 and Q_1 to write its input value into a shared register before enrolling its first integer, we can easily turn agreement on the identity of one of these two processes into agreement on that process' input value. In

[2]dor stands for deterministic one-resilient.

Operations and possible responses: Type dor supports two kinds of operations: ENROLL(v), for $v \in \{0, 1, \ldots, 5\}$, and REVEAL. An ENROLL operation may change the state of the object and always returns the same response, *ack*. A REVEAL operation does not change the state and returns a value, 0 or 1.

States: The type has seventeen states, represented as nodes in the diagram below. The state labeled ☺ is called the *fresh* state; the state labeled ☹ is called the *upset* state; the remaining are called *normal* states.

State transitions: The directed edges of the diagram represent transitions caused by ENROLL operations (as just mentioned above REVEAL operations do not affect the state). The label of an edge denotes the parameter of the corresponding ENROLL. For example, an ENROLL(0) operation applied to the fresh state causes dor to enter the normal state N_0. Note that, in each state, one or more edges (corresponding to one or more ENROLL operations) are missing. The convention regarding missing edges is as follows: Any double-circled node has an implicit transition labeled 5 to itself. Any other missing edge is an implicit transition to the upset state. Thus, for instance, node N_{01} has implicit transitions labeled 0 and 1 to the upset state, and an implicit transition labeled 5 to itself.

Responses: The colour of a node (black or white) indicates the response (0 or 1, respectively) of a REVEAL applied to the corresponding state. Recall that each ENROLL operation returns *ack*, so there is no need to explicitly represent that in the diagram.

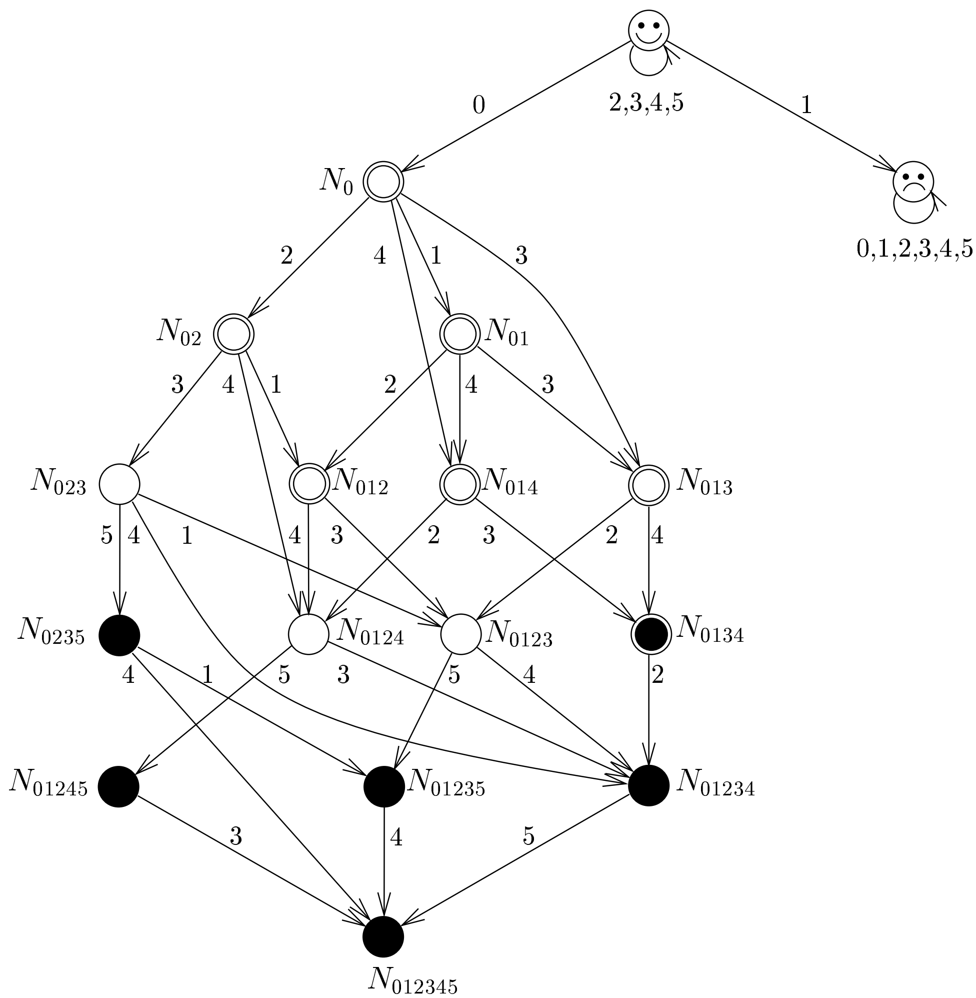

Figure 1: State transition relation of type dor.

this way, we can solve one-resilient Consensus among the three processes using only one dor object and some registers.

We must now make special provisions to ensure that dor cannot be used to implement one-resilient Consensus for *two* processes. Informally, we shall accomplish this by ensuring that any use of a dor object that does not conform to the one described above puts the object in a state from which it is impossible to get "useful" information. In particular, it is impossible to reliably determine whether 0 or 1 was the "winner" (i.e., was enrolled first). The nondeterministic type wor described in [10] did this by defining an "upset" state that the object enters if not accessed as required by the algorithm described above. The object responds by nondeterministically returning 0 or 1 if queried about the winner while in that state. We cannot use this technique here because we want to define a deterministic type. Instead, we allow the type to return deterministic, but misleading, responses if accessed "inappropriately". Doing so while ensuring that the resulting type is not strong enough to implement one-resilient Consensus for two processes turns out to be nontrivial.

We now describe the type dor in more detail. It supports the following operations: ENROLL(v), for $v \in \{0, 1, \ldots, 5\}$, and REVEAL. An ENROLL($v$) is used to enroll v; it always returns the same response, *ack*. A REVEAL is used to find out if 0 was enrolled before 1; it returns either 0 or 1. Intuitively, response 1 is supposed to indicate that 1 was enrolled before 0 (or that neither has been enrolled yet); while response 0 is supposed to indicate that 0 was enrolled before 1. Under certain conditions, however, even though 0 was enrolled before 1, the object will (misleadingly) respond 1 to such a question. This will be the case if the question is asked "prematurely", or after the object has been "mishandled". The question is premature if it is asked before at least two of $\{3, 4, 5\}$ have been enrolled. The object is mishandled if and only if, starting with the enrollment of 0, one of the following conditions takes place: (a) any number other than 5 is enrolled more than once, or (b) 5 is enrolled more than once after both 2 and one of $\{3, 4\}$ have been enrolled.

V-1: Changed this back. Later we say "recall that REVEAL does not change the state, so we need to say this! WK-2: We say "recall that ..." within Figure 1, where we have mentioned this fact. Therefore, I think we don't have to mention it in the main text. To avoid this confusion, I have changed "recall that" to "as just mentioned above" in the description of state transitions in Figure 1.

The behaviour of dor is formally specified by the state transition diagram in Figure 1. To achieve the behaviour described in the preceding paragraph, type dor has seventeen states, consisting of the *fresh* state, denoted ☺, the *upset* state, denoted ☹, and fifteen *normal* states. The fresh state is a source, in the sense that it is not reachable from any other state. The upset state is a sink, in the sense that no other state is reachable from it. Intuitively, it is the state reached if the object is initialised to the fresh state and either 1 is enrolled before 0, or the object is mishandled. The normal states are the states reached if the object is initialised to the fresh state, 0 is enrolled before 1, and the object is not mishandled. In the diagram, each normal state is denoted by a node labeled $N_{v_1 \ldots v_i}$, where $\{v_1, \ldots, v_i\}$ is a nonempty subset of $\{0, 1, \ldots, 5\}$. Intuitively, the subscript of a normal state represents the set of numbers enrolled "so far", assuming the object is initialised to the fresh state. However, some enrollments are "ineffectual" (they do not affect the state), and some enrollments are "implicit" (they occur as a side-effect of other enrollments). Specifically, any attempt to enroll a number before enrolling 0 or 1 is ineffectual. (This corresponds to the self-loop at the fresh state.) Also, any attempt to enroll 5 before both 2 and one of $\{3, 4\}$ have been enrolled is ineffectual. (This corresponds to the double-circled normal nodes.) An implicit enrollment of 1 is forced if an attempt is made to enroll 4 before 1 has been enrolled, or to enroll 3 before both 1 and 2 have been enrolled. (This corresponds to the state transitions that "jump" a level in the diagram — e.g., the transitions labeled 3 and 4 at N_0.) Note that the subscripts of the white normal states in Figure 1 are precisely those that contain at most one of $\{3, 4, 5\}$. In other words, they are precisely the states where at most one of $\{3, 4, 5\}$ has been (effectively) enrolled. Recall, from our earlier informal discussion of type dor, that these are the states in which REVEAL returns a misleading response because the question whether 0 was enrolled before 1 is asked "prematurely".

WK-2: Add three sentences taken from our TR.

Two operations *commute* at a state if applying them in either order results in the same state. For example, ENROLL(2) and ENROLL(1) commute at N_0. An operation *op overwrites op′* at a state, if applying *op* after *op′* at the state results in the same state as applying *op* by itself. For example, ENROLL(3) overwrites ENROLL(1) at N_0. The reader may notice that type dor exhibits these two properties in abundance. The specification of dor was crafted with this goal in mind, as this turns out to be crucial in proving that dor cannot implement one-resilient Consensus between two processes (see Section 2.3).

2.2 Type dor is not too weak ...

In this subsection we prove that dor (together with registers) implements one-resilient Consensus for three processes. An algorithm that does this was described informally in the previous subsection; it is shown in pseudocode in Figure 2. The notation Apply(Q, *op*, O) denotes the procedure by which process Q applies operation *op* to object O; the procedure returns the response of the object to that operation. Register R_k, $k \in \{0, 1, 2\}$, is used by process Q_k to indicate how many numbers it has enrolled so far. Before enrolling

Shared: O: dor, initialised to state ☺
D_0, D_1: register, each initialised to $\perp$
R_0, R_1, R_2: register, each initialised to 0

Code for process Q_k, for $k \in \{0, 1\}$

$D_k :=$ initial value of Q_k
for $\ell := 1, 2$ **do**
    Apply$(Q_k, \textsc{enroll}(3(\ell - 1) + k), O)$
    $R_k := \ell$
    wait until $R_{\overline{k}} \geq \ell$ **or** $R_2 \geq \ell$
$i :=$ Apply$(Q_k, \textsc{reveal}, O)$
decide D_i

Code for process Q_2

for $\ell := 1, 2$ **do**
    wait until $R_0 \geq \ell$ **or** $R_1 \geq \ell$
    Apply$(Q_2, \textsc{enroll}(3(\ell - 1) + 2), O)$
    $R_2 := \ell$
$i :=$ Apply$(Q_2, \textsc{reveal}, O)$
decide D_i

Figure 2: A one-resilient Consensus algorithm for three processes using objects of type dor and registers.

their first numbers, Q_0 and Q_1 write their input values into registers D_0 and D_1, respectively.

Theorem 1 *The algorithm in Figure 2 is a one-resilient Consensus algorithm for three processes that uses only objects of type* dor *and registers.*

2.3 ... nor is it too strong

In this subsection we show that, using only objects of type dor and registers, we cannot solve wait-free Consensus for two processes. (Recall that the concepts of wait-freedom and one-resilience coincide for two-process algorithms.) The argument uses a technique introduced in [10], and is outlined below.

Assume, by way of contradiction, that there exists a wait-free Consensus algorithm $\mathbf{A}$ for processes P_0 and P_1 that uses only objects of type dor and registers. Assume, further, that $\mathbf{A}$ uses a minimal number of dor objects. (This number is finite because the algorithm is wait-free, it uses only deterministic types, and in Consensus each process has only finitely many different input values.) Since wait-free Consensus for two processes is not solvable using only registers [12], it follows that $\mathbf{A}$ uses at least one dor object.

To derive a contradiction, we proceed in two stages. In the first stage, we show that the assumed algorithm $\mathbf{A}$ must have a special configuration (i.e., global state), $\mathcal{B}$, and a dor object, $\mathcal{O}$, with the following properties: For each $k \in \{0, 1\}$, process P_k has an $\textsc{enroll}(k+2)$ operation to $\mathcal{O}$ pending in $\mathcal{B}$. Starting from $\mathcal{B}$, after applying $\textsc{enroll}(k+2)$ to $\mathcal{O}$, P_k cannot apply $\textsc{enroll}(v)$ to $\mathcal{O}$, for any $v \in \{0, 1, 2, 3\}$, and cannot decide unless it has applied

- an $\textsc{enroll}(4)$ to $\mathcal{O}$; or
- an $\textsc{enroll}(5)$ to $\mathcal{O}$, after the other process has taken at least one step.

Furthermore, if each process is about to apply to $\mathcal{O}$ the $\textsc{enroll}(4)$ or $\textsc{enroll}(5)$ that it must execute before deciding, one process must be about to apply $\textsc{enroll}(4)$ while the other must be about to apply $\textsc{enroll}(5)$: they cannot both be about to apply $\textsc{enroll}$ with the same parameter.

We use the bivalence argument of [4] to prove the existence of $\mathcal{B}$ and $\mathcal{O}$. Here is the sketch of proof. (We assume the reader is familiar with the terminology used in the bivalence argument.) First, we show that $\mathbf{A}$ has a (bivalent) critical configuration $\mathcal{C}$ such that each of P_0 and P_1 has an operation to the same dor object $\mathcal{O}$ pending in $\mathcal{C}$. Then, we show that these two pending operations do not commute and neither one of them overwrites the other at the state of $\mathcal{O}$ in $\mathcal{C}$. Using these facts, by an exhaustive case analysis of the states of dor, we show that $\mathcal{O}$ is in the fresh state in $\mathcal{C}$. Finally, we prove that there exists a configuration $\mathcal{B}$ reachable from $\mathcal{C}$ such that $\mathcal{B}$ and $\mathcal{O}$ satisfy all the properties described above.

In the second stage of the proof, starting from configuration $\mathcal{B}$ of the assumed algorithm $\mathbf{A}$ that solves Consensus for processes P_0 and P_1, we show how to solve Consensus for processes Q_0 and Q_1 without using the dor object $\mathcal{O}$. The idea is very similar to the algorithm used in [10]: Each process Q_k simulates the actions of the corresponding process P_k in an execution of $\mathbf{A}$, and uses this execution to determine what value to decide. In this simulation, the processes are allowed to access all objects used by $\mathbf{A}$, except $\mathcal{O}$, each of which is initialised to the state it has in configuration $\mathcal{B}$. More precisely, each process Q_k starts its simulation of P_k by pretending that P_k is in the state it has in configuration $\mathcal{B}$. Based on the current state of (the simulated) P_k, Q_k determines the operation *op* that P_k would apply next, and the object O to which *op* would be applied. If $O \neq \mathcal{O}$, then Q_k has access to O and can apply *op* to it directly to determine the response and update the (simulated) state of P_k accordingly. If, however, $O = \mathcal{O}$, this is not possible, since Q_k is not allowed to access $\mathcal{O}$. Instead, in this case, Q_k acts as follows:

If op is a $\textsc{reveal}$ *operation*, then Q_k pretends that it applied this operation to $\mathcal{O}$ and received response 1.

Shared:

all objects used in **A** except the dor object $\mathcal{O}$, each initialised to the state it has in $\mathcal{B}$

D_0, D_1: register, each initialised to $\perp$

R_0, R_1: register, each initialised to 0

Code for process Q_k, $k \in \{0,1\}$

```
1   D_k := initial value of Q_k
2   state := state of P_k in B
3   while Q_k has not decided do
4       ⟨op, O⟩ := NextOp(P_k, state)
5       if O ≠ 𝒪 then
6           r := Apply(P_k, op, O)
7           state := NextState(P_k, state, r)
8       else (* O = 𝒪 *)
9           if op = REVEAL then
10              state := NextState(P_k, state, 1)
11          else if op = ENROLL(k + 2) then
12              R_k := 1
13              state := NextState(P_k, state, ack)
14          else if op = ENROLL(5) then
15              if R_k̄ = 0 then
16                  state := NextState(P_k, state, ack)
17              else
18                  decide D_k̄
19          else (* op = ENROLL(4) *)
20              decide D_k
```

Figure 3: Solving wait-free Consensus for two processes using **A** with one fewer dor object.

If op is an ENROLL(v) *operation*, then there are three cases:

CASE 1. $v = k+2$: Q_k pretends that it applied this operation to $\mathcal{O}$ and received response *ack*.

CASE 2. $v = 4$: Q_k stops simulating steps of P_k in **A** and decides its own initial value.

CASE 3. $v = 5$: Q_k first checks whether $Q_{\overline{k}}$ has simulated its first step of $P_{\overline{k}}$. If not, Q_k pretends that it applied the ENROLL(5) operation to $\mathcal{O}$ and received response *ack*. Otherwise, Q_k stops simulating steps of **A** and decides $Q_{\overline{k}}$'s initial value.

If Q_k does not decide, then it updates the (simulated) state of P_k as if *op* were applied to $\mathcal{O}$ (as described above) and then proceeds with the simulation of the next operation of P_k.

The algorithm is described more formally in Figure 3. A few remarks about the conventions used in the pseudocode are in order. Recall, from Figure 2, that Apply(P_k, op, O) is the procedure by which P_k invokes operation *op* on object O; it returns the result of this invocation (and updates the state of O accordingly). In addition, we assume that we are given two functions that describe the behaviour of processes P_0 and P_1 in **A**:

- NextOp(P_k, s): returns the pair $\langle op, O\rangle$, where the next operation that P_k executes in **A** when in state s is to apply *op* to object O.
- NextState(P_k, s, r): returns the state that P_k enters in **A** if it receives response r from the operation it invokes when in state s.

In the algorithm, register R_k, $k \in \{0,1\}$, with initial value 0 is used by process Q_k to indicate whether it has simulated a step of P_k. Specifically, Q_k writes 1 into R_k to signify that it has simulated the step of P_k that is pending in $\mathcal{B}$ (line 12). Another shared register D_k is used by Q_k to write its initial value for Consensus before it starts simulating the steps of P_k (line 1).

Roughly speaking, the fact that in algorithm **A**, starting from configuration $\mathcal{B}$, processes P_0 and P_1 must apply ENROLL operations to $\mathcal{O}$ with *different* parameters (namely 4 and 5) before they can decide, is exploited to ensure that the algorithm in Figure 3 satisfies Agreement. Also, the fact that the process that applies ENROLL(5) to $\mathcal{O}$ before deciding must do so in a *nonsolo* execution is exploited to ensure that the algorithm in Figure 3 satisfies Validity. Thus,

Lemma 1 *If* **A** *is a wait-free Consensus algorithm for processes* P_0 *and* P_1*, then the algorithm in Figure 3 is also a wait-free Consensus algorithm for processes* Q_0 *and* Q_1*, using one fewer* dor *object.*

By Lemma 1, the algorithm in Figure 3 solves wait-free Consensus for two processes using one dor object less than **A**. This contradicts our hypothesis that the number of dor objects used in **A** is minimal. Thus,

Theorem 2 *There is no wait-free Consensus algorithm for two processes using only objects of type* dor *and registers.*

3 The case of four or more processes

In this section we show that for $n \geq 4$, any set of types $\mathcal{S}$ (that includes register) strong enough to solve one-resilient Consensus among n processes is also strong enough to solve one-resilient Consensus among $n-1$ processes. This holds even if $\mathcal{S}$ may contain nondeterministic types. The proof of this result is based on a simulation similar to the ones used by Borowsky and Gafni [1] and Chandra *et al.* [3]. Our simulation has some novel aspects, however.

Shared:

Active: **array** $[0..s-1]$ of register,
each is a Boolean variable, initially false
Closed: register, a Boolean variable, initially false
Done: register, a Boolean variable, initially false

TEST&SET. Code for process Q_k, $k \in \{0, \ldots, s-1\}$

```
if Closed then return 1
Closed := true
Active[k] := true
for i := k + 1 to s − 1 do
    repeat
        for j := 0 to k − 1 do
            if Active[j] then
                Active[k] := false; return 1
    until not Active[i]
(* Beginning of CRITICAL SECTION *)
if not Done then
    Done := true
    result := 0
else
    result := 1
(* End of CRITICAL SECTION *)
Active[k] := false; return result
```

Figure 4: A non-wait-free implementation of a TAS register using only registers.

3.1 Type test-and-set-register

The object type test-and-set-register has two states 0 and 1, and supports a single operation TEST&SET. A TEST&SET sets the state to 1 and returns the old state. Thus, the first time TEST&SET is applied to state 0, it changes the state and returns 0; thereafter, any TEST&SET returns 1 leaving the state unchanged. In the sequel we shall refer to an object of type test-and-set-register as a "TAS register".

Our simulation (of a one-resilient Consensus algorithm for $n \geq 4$ processes by only three processes) uses a non-wait-free implementation of test-and-set-register, initialised to state 0, for three processes. The implementation is not wait-free because some correct process may be unable to finish a TEST&SET operation it has applied if some other process crashes while applying its own operation. However, it guarantees that each crash of a process can prevent at most one correct process from finishing its TEST&SET invocation. More precisely, our implementation of TAS registers ensures that in any execution in which processes apply (perhaps repeatedly) TEST&SET operations, the following "limited blocking" property holds: The number of correct processes that start a TEST&SET and never return a response does not exceed the number of processes that start a TEST&SET but crash before completing it. As we shall see later, this property is crucial to our simulation.

An implementation of TAS registers for $s \geq 2$ processes $Q_0, \ldots, Q_{s-1}$ with the limited blocking property is shown in Figure 4. It is based on a mutual exclusion algorithm discovered independently by Burns and Lynch [2], and Lamport [9]. A shared variable *Closed*, initially false, is used to record whether a TEST&SET has already been invoked on the TAS register. To apply a TEST&SET, a process first checks if *Closed* is true (meaning, that a TEST&SET has been invoked). If so, the process immediately returns 1. Otherwise, the process sets *Closed* to true and tries to enter the critical section (CS) — perhaps competing with other processes that also found *Closed* = false. The TEST&SET operation that is the first to gain entry to the CS returns 0; all others return 1. To determine which operation is the first to gain entry to the CS, we use a shared Boolean variable *Done* with initial value false. The first process that enters the CS finds *Done* = false, and sets *Done* to true before it leaves the CS.

The next lemma shows that the algorithm in Figure 4 implements a (non-wait-free) TAS register with the limited blocking property mentioned above.

Lemma 2 *Consider any infinite concurrent execution of processes $Q_0, \ldots, Q_{s-1}$ in which each of them invokes (sequentially) zero or more* TEST&SET *operations by executing the algorithm in Figure 4. In this execution,*

(a) the number of correct processes that do not finish the TEST&SET *invocations they have initiated does not exceed the number of processes that crash;*

(b) if no process crashes, then some process' first TEST&SET *invocation returns 0;*

(c) at most one TEST&SET *invocation returns 0.*

3.2 The simulation algorithm

Suppose we are given an arbitrary one-resilient Consensus algorithm $\mathbf{A}$ for n processes, $P_0, P_1, \ldots, P_{n-1}$, for some $n \geq 4$. We make no assumptions about the set of base objects used by this algorithm. We show how, using $\mathbf{A}$, three processes Q_0, Q_1 and Q_2, can solve one-resilient Consensus. Let $\mathcal{Q} = \{Q_0, Q_1, Q_2\}$ and $\mathcal{P} = \{P_0, P_1, \ldots, P_{n-1}\}$.

The main idea is that the processes in $\mathcal{Q}$ will simulate the steps of the processes in $\mathcal{P}$ so that they end up simulating an execution of $\mathbf{A}$ in which the initial value of every process in $\mathcal{P}$ that has taken steps is equal to that of some process in $\mathcal{Q}$. As soon as some $P \in \mathcal{P}$ decides in the simulated execution, the processes in $\mathcal{Q}$

Shared:

all objects used in **A**, each has the initial state specified by **A**

TAS: **array** $[0..n-1, 1..\infty]$ of test-and-set-register, each implemented by the algorithm in Figure 4

Instr: **array** $[0..n-1]$ of register, each initialised to 1

State: **array** $[0..n-1]$ of register, each initialised to $\perp$

Decision: register, initialised to $\perp$

Code for process Q_k, $k \in \{0, 1, 2\}$

```
i := 0
while Decision = ⊥ do
    r := Instr[i]
    winner := 1
    coexecute
        winner := Apply(Q_k, TEST&SET, TAS[i, r])
        ▮
        repeat until Decision ≠ ⊥
    coend
    if winner = 0 then
        (* Q_k must simulate the rth step of P_i *)
        if r = 1 then   (* first step of P_i *)
            let u be the initial value of process Q_k
            State[i] := initial state of P_i where P_i
            has initial value u
        ⟨op, O⟩ := NextOp(P_i, State[i])
        res := Apply(P_i, op, O)
        State[i] := NextState(P_i, State[i], res)
        if P_i has decided in State[i] then
            Decision := the value decided by P_i
        else
            Instr[i] := Instr[i] + 1
    i := (i + 1) mod n
decide Decision
```

Figure 5: Solving one-resilient Consensus for three processes using **A**.

will adopt P's decision and solve Consensus. Special care must be exercised to ensure that if at most one process in $\mathcal{Q}$ crashes, the simulated execution of **A** will be one in which at most one process in $\mathcal{P}$ crashes. Since **A** is one-resilient, eventually some process will decide in the simulated execution, and thus so will the correct processes in $\mathcal{Q}$.

To ensure that each step of **A** is simulated, the processes in $\mathcal{Q}$ simulate the steps of processes in $\mathcal{P}$ in round-robin fashion. Specifically, each Q_k tries to simulate the first step of P_0, the first step of P_1 and so on, until the first steps of all processes in $\mathcal{P}$ have been simulated. Then Q_k tries to simulate the second step of P_0, the second step of P_1 and so on, until the second steps of all processes in $\mathcal{P}$ have been simulated.

For the simulation to be proper, however, the processes in $\mathcal{Q}$ must coordinate to ensure that they don't repeat the simulation of a step that has already been simulated. (Such a repetition would be disastrous because then the simulated execution might not be a legitimate execution of **A**. Consider, for example, what would happen if an "increment by one" step is simulated several times!) To prevent this, we use TAS registers. For each i, $0 \leq i \leq n-1$, and each $r \geq 1$, there is a TAS register $TAS[i, r]$ initialised to 0, implemented using the algorithm in Figure 4. Before simulating the rth step of process P_i, process Q_k applies a TEST&SET operation to $TAS[i, r]$. If Q_k wins $TAS[i, r]$ (i.e., Q_k's TEST&SET returns 0), it is the unique process responsible for simulating the rth step of P_i. Otherwise, that step has already been or will be simulated by another process in $\mathcal{Q}$, and Q_k goes on to simulate a step of the next process.

By using $TAS[i, r]$ registers implemented as discussed in Section 3.1, we ensure that the crash of one of the processes in $\mathcal{Q}$, in the middle of applying the TEST&SET operation to $TAS[i, r]$, will block at most one other process' execution of TEST&SET on $TAS[i, r]$ (Lemma 2(a)). Since there are three processes in $\mathcal{Q}$, there will still remain a process in $\mathcal{Q}$ which neither crashes nor is blocked by the crash. This process will continue simulating steps of the processes in $\mathcal{P}$ (except for P_i — the process in the simulation of whose step the crash occurred) until one of them decides. (This, incidentally, is the reason why we need at least three simulator processes, and the reason why this simulation cannot be made to work in the case of $n = 3$ — discussed in Section 2.)

The simulation algorithm described above is shown in Figure 5. Besides the $TAS[i, r]$ registers already mentioned, the simulation uses the following shared variables:

- *Instr*: an array $[0..n-1]$ of n registers. $Instr[i]$ contains the index of the first instruction of P_i that has not been simulated yet.
- *State*: an array $[0..n-1]$ of n registers. $State[i]$ contains the current (simulated) state of P_i.
- *Decision*: a register. If a process Q in $\mathcal{Q}$ simulates a step of some process P in $\mathcal{P}$ as a result of which P decides, then Q writes P's decision into *Decision*.

In addition to the notation and conventions introduced in conjunction with the simulation described in Figure 3, we also use the construct "**coexecute** M_1 ▮ M_2 **coend**", where M_1 and M_2 are arbitrary statements (lines 5–8). This interleaves the execution of steps of the two (potentially nonterminating) statements, until one of the two terminates. When (and if) that occurs, the execution of steps of the other statement is abandoned, and the **coexecute** statement itself terminates.

We now explain the reason for using the **coexecute** statement. Suppose a process in $\mathcal{Q}$, say Q_k, crashes while it is executing a TEST&SET on $TAS[i,r]$. The crash of Q_k may cause another process, say Q_ℓ, to get stuck forever in *its* execution of TEST&SET on $TAS[i,r]$. By Lemma 2(a), the remaining process in $\mathcal{Q}$, say Q_m, will not be stuck forever in its execution of TEST&SET on $TAS[i,r]$. Thus, Q_m will be able to simulate steps of processes in $\mathcal{P}$ (other than P_i) until one of them decides. Q_m will then write its decision into *Decision* and after breaking out of the **while** loop (lines 2–20), it will decide. Remember, however, that there is also a correct process in $\mathcal{Q}$, namely Q_ℓ, that is stuck in a nonterminating execution of TEST&SET on $TAS[i,r]$. By applying TEST&SET to $TAS[i,r]$ as one branch of a **coexecute** statement (line 6) and checking whether $Decision \neq \perp$ in another branch (line 7), we ensure that Q_ℓ will eventually break out of the potentially nonterminating TEST&SET invocation, and will decide.

Lemma 3 *If* **A** *is a one-resilient Consensus algorithm for $n \geq 4$ processes $P_0, \ldots, P_{n-1}$ using $\mathcal{S}$, then the algorithm in Figure 5 is a one-resilient Consensus algorithm for three processes Q_0, Q_1, Q_2 using $\mathcal{S}$.*

Since the TAS register implementation of Section 3.1 (Figure 4) uses only registers, by Lemma 3 the simulation in Figure 5 yields a one-resilient Conensus algorithm for three processes that uses the objects used by the Consensus algorithm **A** (i.e., objects of types that belong to $\mathcal{S}$) and registers. Since $\mathcal{S}$ contains the type register, the resulting Consensus algorithm uses only object types that are in $\mathcal{S}$. Thus,

Theorem 3 *Let $\mathcal{S}$ be any set of types (deterministic or nondeterministic) that includes* register. *For any $n \geq 4$, if $\mathcal{S}$ implements Consensus among n processes, then $\mathcal{S}$ implements Consensus among three (and therefore among $n-1$) processes.*

The simulation outlined above uses an unbounded number of TAS registers (one for each potential step of the simulated algorithm). In the full paper we show how to rectify this shortcoming, by using "resettable" TAS registers.

4 Conclusion

The results of this paper, together with those in [3], completely characterise the relationship between the solvability of Consensus among different numbers of processes. More precisely, the results show that, given any set $\mathcal{S}$ of object types and any integers n, t such that $n-1 > t \geq 1$, the statement

> t-resilient Consensus among n processes is solvable using $\mathcal{S}$ *if and only if* t-resilient Consensus among $n-1$ processes is solvable using $\mathcal{S}$

is valid if and only if $t \geq 2$ or $n \geq 4$. Besides this characterisation, these results also reveal a qualitative difference between level one and other levels of the Consensus hierarchy [5, 7]. If $\mathcal{S}$ contains an object type at level two or above of the Consensus hierarchy, the result of Chandra *et al.* implies that the statement above is valid. On the other hand, if $\mathcal{S}$ contains only object types at level one of the Consensus hierarchy, our result in Section 2 shows that the statement is false. This shows that level one of the Consensus hierarchy may have a richer structure than other levels.

Type dor defined in Section 2 is the only deterministic type we know of that is at level one of the Consensus hierarchy and has no one-resilient implementation for three or more processes using only registers.

The simulation sketched in Section 3.2 was first used by Borowsky and Gafni [1], and later by Chandra *et al.* [3]. The main innovation here is the TAS register implementation of Section 3.1. Borowsky and Gafni used a simulation that requires processes to agree on the outcome of each step by solving (a restricted form of) Consensus using only (read/write) registers. Instead of agreeing on the outcome of the step, we use (a similarly restricted form of) TAS registers to ensure that only one process simulates each step. The simulation of Chandra *et al.* applies to a context in which TAS registers are available directly and need not be implemented.

Acknowledgements

Our thinking on shared memory distributed computing has been deeply influenced from discussions with Tushar Chandra, Prasad Jayanti and Sam Toueg. We are grateful to Faith Fich for her helpful comments on earlier drafts of this paper.

References

[1] Elizabeth Borowsky and Eli Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *Proceedings of the Twenty-Fifth ACM Symposium on Theory of Computing*, pages 91–100, May 1993.

[2] James Burns and Nancy Lynch. Mutual exclusion using indivisible reads and writes. In *Proceedings of Eighteenth Annual Allerton Conference on Communications, Control and Computing*, pages 833–842, 1980.

[3] Tushar Chandra, Vassos Hadzilacos, Prasad Jayanti, and Sam Toueg. Wait-freedom versus t-resiliency and the robustness of wait-free hierarchies. In *Proceedings of the Thirteenth ACM Symposium on Principles of Distributed Computing*, pages 334–343, August 1994.

[4] 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.

[5] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, January 1991.

[6] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[7] Prasad Jayanti. On the robustness of Herlihy's hierarchy. In *Proceedings of the Twelfth ACM Symposium on Principles of Distributed Computing*, pages 145–158, August 1993.

[8] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[9] Leslie Lamport. The mutual exclusion problem: Parts I & II. *Journal of the ACM*, 33(2):313–348, 1986.

[10] Wai-Kau Lo. More on t-resilence vs. wait-freedom. In *Proceedings of the Fourteenth ACM Symposium on Principles of Distributed Computing*, pages 110–119, August 1995.

[11] Wai-Kau Lo and Vassos Hadzilacos. On the power of shared object types to implement one-resilient consensus. Technical Report CSRI-357, University of Toronto, Toronto, Ontario, Canada, January 1997. Available at ftp://ftp.cs.utoronto.ca.pub/reports/csri/357.

[12] Michael Loui and Hosame Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. In *Advances in Computer Research*, volume 4, pages 163–183. JAI Press Inc., 1987.

[13] Michael Stumm and Songnian Zhou. Fault tolerant distributed shared memory. In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing*, pages 719–724, December 1990.