m-Consensus Objects Are Pretty Powerful

Ammar Qadri

University of Toronto, Canada
ammar.qadri@mail.utoronto.ca

Abstract

A recent paper by Afek, Ellen, and Gafni introduced a family of deterministic objects $O_{m,k}$, for $m, k \geq 2$, with consensus numbers m such that, for each $k \geq 2$, $O_{m,k}$ is computationally less powerful than $O_{m,k+1}$ in systems with at least $mk+m+k$ processes. This paper gives a wait-free implementation of $O_{m,k}$ from $(m+1)$-consensus objects and registers in systems with any finite number of processes. In order to do so, it introduces a new family of objects which helps us to understand the power of m-consensus among more than m processes.

1998 ACM Subject Classification E.1 Distributed Data Structures, F.1.2 Parallelism and concurrency

Keywords and phrases Deterministic Consensus Hierarchy, Wait-free Implementation, Tournament

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

The consensus problem is a fundamental problem in distributed computing that has long been used to categorize the computational powers of shared objects. Herlihy [5] defined the *consensus number* of an object, which is the largest number of processes for which wait-free consensus can be achieved using only instances of the object and registers. Thus an object O with consensus number n cannot be implemented in a wait-free manner by an object O' with consensus number $n' < n$ in a system with more than n' processes. The *consensus hierarchy* classifies objects by their consensus numbers.

Herlihy also proved that any object can be implemented by n-consensus objects and registers (and, hence, by any objects with consensus number n) in systems with n or fewer processes. However, the relative computational powers of objects with the same consensus number n in systems of more than n processes is not entirely understood.

Afek, Gafni, Tromp, and Vitányi [3] showed that test-and-set objects, which have consensus number 2, can be implemented from 2-consensus objects and registers in a system with any finite number of processes. Afek, Weisberger, and Weisman [4] and Afek, Gafni, and Morrison [2] proved that this is also true for other well-known objects of consensus number 2, such as fetch-and-increment objects, swap objects, and stacks. It was conjectured [4] that this is true for any object with consensus number 2. This is known as the *Common2 Conjecture*. More generally, the *Consensus Hierarchy Conjecture* asserts that for $n \geq 2$, every shared object of consensus number $n' \leq n$ has a wait-free implementation from n-consensus objects and registers in every system with a finite number of processes.

Rachman [6] disproved the Consensus Hierarchy Conjecture for nondeterministic objects. He showed that for any m' and m, with $m' \geq m \geq 1$, there exists a nondeterministic object X with consensus number m, such that X cannot be implemented using only m'-consensus objects and registers in systems with at least $2m'+1$ processes. This means that the consensus hierarchy, at least on its own, is not very useful in characterizing the computational powers of nondeterministic objects.

Afek, Ellen, and Gafni [1] proved that the Consensus Hierarchy Conjecture does not even hold for the class of deterministic objects. They introduced the $O_{m,k}$ object, for $m, k \geq 2$, and showed that each $O_{m,k}$ object has consensus number m, but cannot be implemented (in a non-blocking manner) from m-consensus objects and registers in any system with at least $km + k - 1$ processes. More surprisingly, they showed that an $O_{m,k+1}$ object cannot be implemented (in a non-blocking manner) from $O_{m,k}$ objects and registers in any system with at least $mk + m + k$ processes, meaning that $O_{m,2}, O_{m,3}, \ldots$ is an infinite sequence of objects with increasing computational power, all with consensus number m.

This paper determines the relationship between $O_{m,k}$ objects and $(m + 1)$-consensus objects. Section 4 presents a wait-free implementation of every $O_{m,k}$ object from $(m + 1)$-consensus objects and registers among any finite number of processes. Thus, $O_{m,k}$ objects lie strictly between m-consensus objects and $(m+1)$-consensus objects in terms of computational power. This provides additional understanding of the consensus hierarchy for deterministic objects and is a step towards a characterization of their computational power.

For our implementation, we introduce a new family of deterministic objects, Q_r, for $r \geq 0$, which serves as a crucial synchronization mechanism. The Q_r object is a generalization of the test-and-set object: it has an operation which allows the first process that performs it to win and all subsequent processes to fail. In addition, there is another operation that returns the identity of the winner to the first r processes who perform it after the object has been won. Section 3 formally defines the Q_r object, proves that it has consensus number $r + 2$, and proves that it has a wait-free implementation from $(r + 2)$-consensus objects and registers in any system with a finite number of processes. It is hoped that the Q_r object will be a useful tool for determining the power of $(r + 2)$-consensus objects in systems of more than $r + 2$ processes.

2 Model

We consider an asynchronous shared memory system, with n processes, $p_1, p_2, \ldots, p_n$, which communicate by applying operations to shared objects. A *step* by any process consists of an operation it is performing on a shared object as well as the response received from the operation.

A *configuration* of the shared memory system consists of the current state of every process, as well as the current value of every shared object. An *initial configuration* is a configuration where all processes and shared objects are in one of their initial states. An *execution* is specified by an alternating sequence of configurations and steps by processes, beginning at an initial configuration, such that if a configuration C is immediately followed by a step s of process p_i, which is immediately followed by a configuration C', then p_i performing step s in configuration C yields the configuration C'.

An *implementation* of a sequentially specified shared object O from a set of shared base objects is given by providing an algorithm for each operation of O in which processes only access the base objects. The implementation is *linearizable* if, in any execution $\mathcal{E}$, we can order each completed operation as well a subset of incomplete operations on O such that the results of each operation in this order are consistent with the sequential specification of O, and, moreover, an operation that completes before another operation begins comes earlier in the order. Equivalently, the implementation is linearizable if, in any execution $\mathcal{E}$, we can assign a (distinct) linearization point to each completed operation as well as a subset of incomplete operations such that the linearization point of every operation is within the execution interval of that operation and the results of each operation in the order induced by the linearization points are consistent with the sequential specification of O. The implementation is *wait-free* if every process that performs any operation of O completes the operation in a finite number of its own steps. The implementation is *non-blocking* if, from every configuration of any execution, there is some process that completes its operation in a finite number of steps.

Consensus can be solved by n processes, $p_1, p_2, \ldots, p_n$, if there exists an algorithm such that, if every process p_i is assigned some input x_i, each process outputs at most one value, satisfying the following conditions:

- *validity*: If p_i outputs y_i, then there exists some j such that $y_i = x_j$.
- *agreement*: If p_i outputs y_i and p_j outputs y_j, then $y_i = y_j$.
- *termination*: Every process that takes sufficiently many steps outputs a value.

The m-consensus object is a shared object with a single operation PROPOSE that takes one argument. The first m PROPOSE operations to an m-consensus object return the argument of the first PROPOSE operation. We say that this value is the *decision* of the m-consensus object. All other operations return $\perp$.

The test-and-set object is a shared object with a single operation T&S, which returns *true* for the first operation and *false* for all subsequent operations.

The fetch-and-increment object is a shared object and has a single operation F&I, which returns the value $a + i - 1$ to the i^{th} operation, where a was the initial value of the object.

Throughout this paper, we assume that the initial value of every fetch-and-increment object is 1 and the initial value of every register is $\perp$.

3 The Q_r object

We now formally define the Q_r object and explain how it can be used together with registers to solve consensus among $r + 2$ processes. Then, in Section 3.1, we prove that it can be implemented from $(r + 2)$-consensus objects and registers in a system with any finite number of processes.

The Q_r object, for $r \geq 0$, has two operations, COMPETE and QUERY, with the following specifications:

- The first COMPETE operation returns *true* and the process that performs it is called the *winner* of the object. All subsequent COMPETE operations return $false$.
- The first r QUERY operations after the first COMPETE operation return the *id* of the winner. All other QUERY operations return $\perp$.

If processes perform only COMPETE operations, then a Q_r object behaves like a test-and-set object. In particular, Q_0 is equivalent to a test-and-set object and, thus, has consensus number 2. The additional power of a Q_r object for $r > 0$ is due to the r QUERY operations that can be used to determine the winner.

The Q_r object can solve consensus among $r + 1$ processes: Each process announces its input in a single writer register before calling COMPETE on a shared Q_r object. The winner decides its own input, whereas the r losers call QUERY to find the identity of the winner, and then read and decide the winner's announced value.

It is only slightly more difficult to see that the Q_r object can solve consensus among $r + 2$ processes. As before, each of the processes, $p_1, p_2, \ldots, p_{r+2}$, first announces its input in a single writer register. Next, processes perform COMPETE on a Q_r object and the winner decides its own input. The problem is that there are $r+1$ processes that are not the winner of the Q_r object and may need to learn the identity of the winner. However, the Q_r object can only return the identity of the winner to at most r processes. To resolve this, each process that loses its COMPETE operation first overwrites its announced value with $\perp$ before calling QUERY on the Q_r object. If its QUERY operation is successful, then, as before, it reads and decides the announced value of the winner. However, if its QUERY operation was unsuccessful, then $r+1$ processes have already called QUERY on the Q_r object by this point. Prior to calling QUERY, these processes overwrote their announced values. Thus, exactly one value remains in the announcement array: the value of the winner. The code is provided in Algorithm 1.

Algorithm 1 A solution to $(r+2)$-consensus

```
Shared variables:
$q$ is a $Q_r$ object
$announce[1 \dots r+2]$ is an array of registers

p1: function PROPOSE($v$) by process $p_i$ for $i \in \{1, \dots, r+2\}$
p2:     $announce[i]$.WRITE($v$)
p3:     if $q$.COMPETE() then
p4:         return $v$
p5:     end if
p6:     $announce[i]$.WRITE($\perp$)
p7:     $winner \leftarrow q$.QUERY()
p8:     if $winner \neq \perp$ then
p9:         return $announce[winner]$.READ()
p10:    end if
p11:    for $j \leftarrow 1$ to $r+2$ do
p12:        $value \leftarrow announce[j]$.READ()
p13:        if $value \neq \perp$ then
p14:            return $value$
p15:        end if
p16:    end for
p17: end function
```

▶ **Observation 1.** *Algorithm 1 solves the consensus task among $r+2$ processes*

▶ **Lemma 2.** *The Q_r object has consensus number at least $r+2$.*

3.1 An implementation from $(r+2)$-consensus objects

We now give a wait-free implementation of a Q_r object shared among any finite number n of processes using only $(r+2)$-consensus objects and registers. In particular, this shows that the Q_r object has consensus number at most $r+2$.

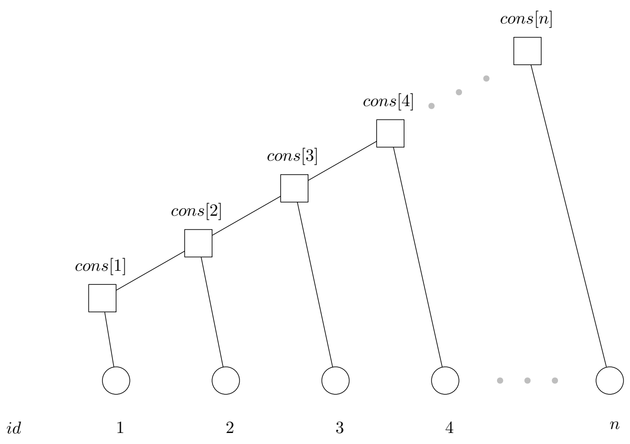
Our implementation uses an array $cons[1 \dots n]$ of $(r+2)$-consensus objects, a register *gate*, and a fetch-and-increment object *count*. Note that fetch-and-increment objects have a wait-free implementation from 2-consensus objects (and, hence, from $(r+2)$-consensus objects, since $r \geq 0$) and registers among any finite number of processes [4].

Processes that call COMPETE participate in a tournament to decide which operation returns *true*. However, we must prevent the situation where, after a COMPETE operation returns $false$, another COMPETE operation begins and returns $true$. No linearization of such an execution would be consistent with the sequential specifications of COMPETE.

To prevent this issue, a process p_i performing COMPETE first checks the value of the shared register $gate$. If $gate \neq \perp$, the COMPETE operation returns $false$. Otherwise, p_i writes its own id, i, to $gate$. This ensures all processes calling COMPETE that write their id to $gate$ are concurrent. This idea was used in the wait-free implementation of an n-process test-and-set object from 2-process test-and-set objects and registers [3].

After process p_i writes to $gate$, it continues by entering a tournament. It proposes i to $cons[i]$ through $cons[n]$ in order. If it receives a response other than i from one of these consensus objects, it immediately returns false and does not propose i to any further consensus objects. If the responses it receives from $cons[i]$ through $cons[n]$ are all i, then p_i returns $true$. The tournament is depicted in Figure 1. Note that $cons[1]$ is only used for simplifying the code and proof of correctness. It is otherwise unnecessary.

Figure 1 Tournament to determine winner



To perform QUERY, a process p_i first reads the value i' in $gate$. If $i' = \perp$, then no process performing COMPETE has written its id to $gate$ and taken part in the tournament, so p_i returns $\perp$. If $i' \neq \perp$, p_i performs F&I on $count$ and, if the value returned is greater than r, returns $\perp$. Note that at most r QUERY operations proceed past this point. If a COMPETE operation has already returned $true$, p_i could just propose to $cons[n]$ to get the id of the winner. However, this is not guaranteed. The competing processes may be slow or the winning process may have crashed before proposing to $cons[n]$. Thus, p_i attempts to assist participants in the tournament by propagating their ids. It begins by proposing i', the id it read from $gate$, to $cons[i']$. Then it proposes the decision of $cons[j-1]$ to $cons[j]$ for $i' < j \leq n$ in order. Finally, it returns the decision of $cons[n]$, which is the id of the winner.

The code for COMPETE and QUERY is presented in Algorithm 2.

We now prove the correctness of the implementation of a Q_r object given in Algorithm 2. Let $\mathcal{E}$ be any execution of the implementation.

We will say that a COMPETE or QUERY *operation* performs a step σ in the execution if the process performing the operation executes σ during the operation.

We consider three cases.

In the first case, suppose that no operation in execution $\mathcal{E}$ ever writes to $gate$. That is, suppose no COMPETE operation executes line c5 during execution $\mathcal{E}$. Then no COMPETE operation returns, since the value of $gate$ remains $\perp$ throughout execution $\mathcal{E}$. Thus, line c3 is not executed, nor are lines c8 or c11. All COMPETE operations are therefore incomplete and

Algorithm 2 An implementation of Q_r

Shared variables:
cons$[1 \ldots n]$ is an array of $(r+2)$-consensus objects
gate is a register initialized to $\perp$
count is a fetch-and-increment object initialized to 1

```
c1:  function COMPETE( ) by process $p_i$
c2:      if $gate$.READ() $\neq \perp$ then
c3:          return $false$
c4:      end if
c5:      $gate$.WRITE($i$)
c6:      for $j \leftarrow i$ to $n$ do
c7:          if $cons[j]$.PROPOSE($i$) $\neq i$ then
c8:              return $false$
c9:          end if
c10:     end for
c11:     return $true$
c12: end function
```

```
q1:  function QUERY( )
q2:      $i' \leftarrow gate$.READ()
q3:      if $i' = \perp$ then
q4:          return $\perp$
q5:      end if
q6:      if $count$.F&I() $> r$ then
q7:          return $\perp$
q8:      end if
q9:      $start \leftarrow i'$
q10:     for $j \leftarrow start$ to $n$ do
q11:         $i' \leftarrow cons[j]$.PROPOSE($i'$)
q12:     end for
q13:     return $i'$
q14: end function
```

we do not linearize any of them. All QUERY operations that complete return $\perp$ on line q4. We can define the linearization point of each of these QUERY operations to be the time at which the operation executes line q2. Since all linearized QUERY operations return $\perp$ and no COMPETE operation has been linearized, this linearization is consistent with the sequential specifications of a Q_r object.

In the second case, suppose that line c5 is executed by some operation in execution $\mathcal{E}$, but no value is ever proposed to any consensus object. That is, neither line c7 nor line q11 is executed by any operation in $\mathcal{E}$. Then let C be the COMPETE operation during which the first write to *gate* takes place and let p_w be the process that performs operation C. Define the linearization point of operation C to be the step at which it writes to *gate* in line c5. In this execution, every COMPETE operation that completes returns $false$ on line c3. For every such completed COMPETE operation, define its linearization point to be the step at which it reads from *gate* in line c2. It follows that for every completed COMPETE operation C', operation C is linearized before operation C', since the read operation performed by C' in line c2 returned a value other than $\perp$, meaning that the first write to *gate*, in line c5 of C, had already taken place.

We also linearize the QUERY operations as follows: Every QUERY operation that returns $\perp$ on line q4 is linearized at the point it executes the read of *gate* in line q2. Thus, every such operation Q is linearized before C, since Q reads $\perp$ from *gate*. The QUERY operations that perform F&I on *count* are linearized at the point they execute line q6. Each of these QUERY operations is linearized after C, since it reads a value other than $\perp$ from *gate* in line q2. Moreover, if one of these QUERY operations Q returns $\perp$, it does so after *count* has already been incremented r times by r other QUERY operations that executed line q6. So there at least r QUERY operations linearized after C and before every such operation Q. This linearization is consistent with the sequential specifications of a Q_r object.

Finally, we consider any execution $\mathcal{E}$ where some value is proposed to a consensus object on line c7 or q11 during some operation in $\mathcal{E}$. Processes are referred to as *competitors* while they execute lines c5–c11 and as *queriers* while they execute lines q9–q13. Note that only competitors and queriers access consensus objects, and there are at most r queriers.

▶ **Observation 3.** *A process can only be a competitor in its first invocation of* COMPETE*, so every competitor has a distinct id.*

We say that i *is accepted by* a consensus object if it is the decision of the consensus object. If i is proposed, but is not accepted, we say it *is rejected from* the consensus object.

Competitor p_i only proposes its *id*, i, to $cons[j]$ if $j = i$ or if i was accepted by $cons[j-1]$. Every querier begins by reading some competitor's *id*, i', in *gate*. It proposes this value i' to $cons[i']$ in the first iteration of the for loop in line q10. In every subsequent iteration, it proposes the value accepted by $cons[j-1]$ to $cons[j]$. This gives us the following observation:

▶ **Observation 4.** *For any $j > 1$, the only possible values proposed to $cons[j]$ are j and the value accepted by $cons[j-1]$. The only possible value proposed to $cons[1]$ is 1.*

From Observation 4, we get the following useful results:

▶ **Lemma 5.** *If $i < j$ is proposed to $cons[j]$, then i was accepted by $cons[\ell]$, for all $i \leq \ell < j$.*

Proof. Suppose not. Consider the largest $\ell < j$ such that i was not accepted by $cons[\ell]$. If $\ell < j - 1$, this means that i was accepted by $cons[\ell + 1]$ and, hence, from the semantics of PROPOSE, was proposed to $cons[\ell + 1]$. If $\ell = j - 1$, then i was proposed to $cons[\ell + 1]$ by assumption. In either case, since $i < l + 1$, it follows from Observation 4 that the value i was accepted by $cons[\ell]$. This is a contradiction. ◀

▶ **Lemma 6.** *No process that proposes to the $(r + 2)$-consensus object $cons[j]$ receives a response of $\perp$.*

Proof. By Observation 4, there are at most two distinct values proposed to $cons[j]$. Since every competitor only proposes its own *id*, by Observation 3, we conclude that there are at most 2 competitors that propose to $cons[j]$. No competitor proposes to $cons[j]$ more than once. Additionally, there are at most r queriers and each querier proposes to $cons[j]$ at most once. It follows that there are at most $r + 2$ PROPOSE operations performed on $cons[j]$ and, hence, none of them receive a response of $\perp$. ◀

By examining the code, we can learn the following fact about the values proposed to consensus objects:

▶ **Observation 7.** *The value i is only proposed to $cons[j]$ if there exists a competitor with id i that has already written i to gate on line c5.*

We will now show that execution $\mathcal{E}$ is linearizable.

Recall that during execution $\mathcal{E}$, some value is proposed to some consensus object. Consider the smallest j such that some value is proposed to $cons[j]$. By Observation 4, the only value proposed to $cons[j]$ is j. It follows that j is accepted by $cons[j]$. Let w be the greatest value such that w is accepted by $cons[w]$ in execution $\mathcal{E}$. Let C_w be the first COMPETE operation of process p_w. By Observations 3 and 7, C_w exists and writes w to *gate* in line c5 (after reading $\perp$ from *gate* in line c2). Thus, operation C_w is concurrent with the first write to *gate* and we linearize it at that point. Every COMPETE operation that returns on line c3 is linearized at the point it executes line c2 and every COMPETE operation that returns on line c8 is linearized at the point of its preceding proposal to a consensus object in line c7.

We now show that C_w is the only COMPETE operation that can return on line c11. This means that we have linearized every completed COMPETE operation.

▶ **Lemma 8.** *If i is accepted by $cons[n]$, then $i = w$.*

Proof. Since i is accepted by $cons[n]$, then, by Lemma 5, it is also accepted by $cons[\ell]$, for all $i \leq \ell < n$. Thus i was accepted by $cons[i]$ and, for any $j > i$, j was not accepted by $cons[j]$. So $i = w$. ◀

▶ **Corollary 9.** *The only* COMPETE *operation that can return true is operation C_w.*

Proof. If COMPETE operation C by process p_i returns *true*, then i is accepted at $cons[n]$ so, by Lemma 8, $i = w$. By Observation 3, only the first COMPETE operation of any process can return *true*, so $C = C_w$. ◀

▶ **Lemma 10.** *Any linearized* COMPETE *operation $C \neq C_w$ is linearized after C_w.*

Proof. Consider any such operation C. Since $C \neq C_w$ is linearized, C has completed and so it returns on line c3 or line c8 by Corollary 9. If operation C returns on line c3, it is linearized when it reads some value other than $\perp$ from *gate* in line c2. Since C_w is linearized at the earliest write to *gate* in execution $\mathcal{E}$, C_w is linearized before C. On the other hand, if operation C returns on line c8, then it performs a write to *gate* in line c5, which is at or after the linearization point of C_w. Operation C is linearized later, when it performs its last proposal to a consensus object in line c7. Thus, C_w is linearized before C. ◀

It follows that p_w is the winner in the linearization of execution $\mathcal{E}$. From Corollary 9, we know that C_w is the only operation that can return *true*. We must also argue that it does not return $false$:

▶ **Lemma 11.** *Operation C_w does not return $false$.*

Proof. For COMPETE operation C_w to return $false$, w must be rejected from $cons[j]$, for some $j > w$. Consider any such j. By Observation 4, w was accepted by $cons[j-1]$ and j was the only other value proposed to $cons[j]$. Since w was rejected from $cons[j]$, it follows that j was accepted by $cons[j]$. But this contradicts the definition of w. ◀

Finally, we can show that it is possible to linearize the QUERY operations. As in the second case, every QUERY operation that returns on line q4 is linearized at the point it reads $\perp$ from *gate* in line q2. This is before the first write to gate, which is when C_w is linearized. Every QUERY operation that calls F&I on *count* in line q6 is linearized at the point it executes line q6, which occurs after it reads a value other than $\perp$ from *gate* in line q2 and, thus, after the linearization point of C_w. Note that all completed QUERY operations either return on q4 or execute line q6. By the semantic of F&I, the QUERY operations that execute line q6 are linearized in increasing order of the values they receive from *count*. In particular, if some such QUERY operation receives a value greater than r, at least r other QUERY operations have been linearized before it (and after C_w).

▶ **Corollary 12.** *Every* QUERY *operation that receives a value of r or less from count can only return the id of the winner p_w.*

Proof. Any such QUERY operation can only return in line q13 and it returns the value accepted by $cons[n]$, which, by Lemma 8, is w. ◀

Thus, the linearization of execution $\mathcal{E}$ is consistent with the sequential specifications of a Q_r object. Since $\mathcal{E}$ was chosen arbitrarily, every execution of Algorithm 2 is linearizable. Note that the implementations of COMPETE and QUERY in Algorithm 2 are wait-free. Therefore, we have shown the following theorem:

▶ **Theorem 13.** *For $r \geq 0$, Q_r has consensus number $r+2$ and can be implemented in a wait-free manner from $(r+2)$-consensus objects and registers in a system with any finite number of processes.*

4 Implementing $O_{m,k}$ from $(m+1)$-consensus objects

In this section, we present the formal definition of the $O_{m,k}$ object from [1], followed by a description of a natural, but incorrect, approach for implementing the $O_{m,k}$ object from $(m+1)$-consensus objects and registers for any number of processes. We then give our wait-free implementation and prove that it is correct.

4.1 Sequential Specifications of $O_{m,k}$

The $O_{m,k}$ object, for $m, k \geq 2$, has a single operation SUGGEST, which takes a non-negative argument, and has the following specifications, where, for $j \in \{1, \ldots, k\}$, a_j is the argument of the $(j-1)m + 1^{st}$ SUGGEST operation:

- For $j \in \{1, \ldots, k\}$, the $(j-1)m + 1^{st}$ through jm^{th} SUGGEST operations return a_j.
- For $j \in \{1, \ldots, k-1\}$, the $km + j^{th}$ SUGGEST operation returns a_{k-j}.
- For $j > km + k - 1$, the j^{th} SUGGEST operation returns $\perp$.

The first km SUGGEST operations performed on an $O_{m,k}$ object are called *prefix operations* and the next $k-1$ SUGGEST operations are called *suffix operations*. A more intuitive way to visualize the behaviour of SUGGEST is with the string $S_{m,k} = A_1^m A_2^m \ldots A_k^m A_{k-1} A_{k-2} \ldots A_1$. For $1 \leq j \leq km + k - 1$, if A_g is the j^{th} character in $S_{m,k}$, then the j^{th} SUGGEST operation returns a_g, and we say it *belongs to group g*. For $j > km + k - 1$, the j^{th} SUGGEST operation returns $\perp$.

4.2 An incorrect approach

A simple algorithm, due to Faith Ellen, for implementing $O_{m,2}$ from $(m+1)$-consensus objects and registers for any number of processes is as follows:

- Each time a process p_i performs SUGGEST(v), it first accesses a fetch-and-increment object to receive a distinct value x.
- If $x > 2m + 1$, then p_i returns $\perp$.
- If $1 \leq x \leq m$, then p_i proposes v to an $(m+1)$-consensus object C_1. It writes the response to a shared register R_1 before returning it.
- If $m + 1 \leq x \leq 2m$, then p_i proposes v to an $(m+1)$-consensus object C_2. It writes the response to a shared register R_2 before returning it.
- If $x = 2m + 1$, p_i reads R_1 and then R_2. It returns the first non-$\perp$ value it reads. If both R_1 and R_2 are $\perp$, this means that all processes that received values at most $2m + 1$ from the fetch-and-increment object are concurrent. In this case, p_i proposes v to C_1, and returns the response.

This algorithm correctly implements an $O_{m,2}$ object from $(m+1)$-consensus objects registers. For each of the consensus objects C_1 and C_2, the SUGGEST operation that first proposes a value to the consensus object can be linearized before the rest of the SUGGEST operations that propose to it.

However, this approach does not scale to $k > 2$. Consider the natural extension of this algorithm to $O_{m,3}$:

- Once again, each time a process p_i performs SUGGEST(v), it first accesses a fetch-and-increment object to receive a distinct value x.
- If $x > 3m + 2$, then p_i returns $\perp$.
- If $(j - 1)m + 1 \leq x \leq jm$ for $j \in \{1, 2, 3\}$, then p_i proposes v to an $(m + 1)$-consensus object C_j. It writes the response to a shared register R_j before returning it.
- If $x \in \{3m + 1, 3m + 2\}$, then p_i will propose to the consensus object associated with some group and return the response.

The problem with the above approach arises from the operations that receive $3m + 1$ and $3m + 2$ from the fetch-and-increment object. Firstly, these operations must propose to the consensus objects associated with different groups. This can be achieved as in [1] by using test-and-set objects. The more difficult issue is ensuring linearizability:

Consider an execution in which some process p_i receives $3m + 1$ from the fetch-and-increment object, reads $R_1 = R_2 = R_3 = \perp$, proposes v to some consensus object C_j, and returns the decision. Next, the m operations that received $(j - 1)m + 1, \ldots, jm$ from the fetch-and-increment object complete their operations, writing to R_j and returning the decision of C_j. Next, another process p_ℓ begins its SUGGEST operation with an argument that is different from the arguments of all preceding SUGGEST operations and receives $3m + 2$ from the fetch-and-increment object. Note that since an entire group (of $m + 1$ SUGGEST operations) has returned before p_ℓ began, p_ℓ must be performing a suffix operation when this execution is linearized. In particular, it cannot return the argument of its own SUGGEST operation.

If the other prefix operations have not made any progress (i.e. they have not taken any steps since receiving a value from the fetch-and-increment object), then p_ℓ cannot determine any of their arguments, unless it waits for them, which it cannot do. Even if each SUGGEST operation announces its argument before it accesses the fetch-and-increment object, p_ℓ does not know which prefix operations belong to the same group as p_i's operation and, so, cannot determine a value to return.

4.3 A wait-free implementation

The following implementation starts by assigning m prefix operations to each of the k groups, and then assigns the $k - 1$ suffix operations to the first $k - 1$ groups. It ensures that suffix operations can determine and propose the argument of some prefix operation in their group. To do this, we require a stronger synchronization mechanism than a fetch-and-increment object: namely, the Q_1 object.

To implement the $O_{m,k}$ object in a system with any finite number of processes, n, from $(m + 1)$-consensus objects and registers, we use an array *cons*$[1 \ldots k]$ of $(m + 1)$-consensus objects and an array *position*$[1 \ldots km]$ of Q_1 objects. Note that since the Q_1 object can be implemented by 3-consensus objects and registers for any finite number of processes and $m + 1 \geq 3$, $(m + 1)$-consensus objects and registers can also implement the Q_1 object for any finite number of processes.

A process p_i performing SUGGEST(v) will perform COMPETE on objects *position*$[1]$ through *position*$[km]$ in order, until it wins one of them or loses all km of them.

If p_i is the winner of the Q_1 object *position*$[j]$, it is performing a prefix operation. It will propose v to the consensus object *cons*$[\lceil \frac{j}{m} \rceil]$ associated with its group and return its response.

If p_i fails to win any Q_1 object, it is either a suffix operation or it returns $\perp$. It accesses a fetch-and-increment object *count* that is initially 1. Note that fetch-and-increment objects have a wait-free implementation for any finite number of processes from 2-consensus objects and registers and, consequently, since $m+1>2$, also from $(m+1)$-consensus objects and registers. Let x be the response from *count* that p_i receives. If $x > k-1$, p_i returns $\perp$. Otherwise, it is performing the suffix operation associated with group $k-x$. It performs QUERY on $position[(k-x)m]$ to get the identity i' of some process that performed a prefix operation associated with group $k-x$. If each process announces the argument of its SUGGEST operation at the beginning of the operation, then p_i could read the value announced by $p_{i'}$, propose this value to $cons[k-x]$, the consensus object associated with its group, and return the response. This ensures that the arguments of prefix operations are the only non-$\perp$ values returned.

There is a slight problem with this approach. Process $p_{i'}$ could have performed another SUGGEST operation afterwards that overwrote the value it announced before winning $position[(k-x)m]$. Instead, we use an array $A_j[1\ldots n]$ of n registers associated with each $position[j]$. Before performing a COMPETE operation on $position[j]$, each process p_ℓ writes the argument of its current SUGGEST operation to $A_j[\ell]$, provided it has not previously written there. Then process p_i can read $A_{(k-x)m}[i']$ to find the argument of a prefix operation in group $k-x$.

The code for the implementation is given in Algorithm 3.

Algorithm 3 An implementation of $O_{m,k}$

Shared variables:
$cons[1\ldots k]$ is an array of $(m+1)$-consensus objects
$A_j[1\ldots n]$ is an array of registers initialized to $\perp$, for $j \in \{1,\ldots,km\}$
$position[1\ldots km]$ is an array of Q_1 objects
$count$ is a fetch-and-increment object initialized to 1

```
s1:  function SUGGEST($v$) by process $p_i$
s2:      for $j \leftarrow 1$ to $km$ do
s3:          if $A_j[i]$.READ() $= \perp$ then
s4:              $A_j[i]$.WRITE($v$)
s5:              if $position[j]$.COMPETE() then
s6:                  return $cons[\lceil \frac{j}{m} \rceil]$.PROPOSE($v$)
s7:              end if
s8:          end if
s9:      end for
s10:     $x \leftarrow count$.F&I()
s11:     if $x > k-1$ then
s12:         return $\perp$
s13:     end if
s14:     $group \leftarrow k-x$
s15:     $member \leftarrow position[group \times m]$.QUERY()
s16:     $value \leftarrow A_{group\times m}[member]$
s17:     return $cons[group]$.PROPOSE($value$)
s18: end function
```

4.4 Correctness

We will show that the implementation of an $O_{m,k}$ object given in Algorithm 3 is linearizable. Consider any execution $\mathcal{E}$ of this implementation.

We will say that a SUGGEST *operation* performs a step σ in the execution if the process performing the operation executes σ during the operation.

A SUGGEST operation S *fills* the Q_1 object *position*$[j]$ if it performs the COMPETE operation on *position*$[j]$ that returns *true*. In this case, operation S will be ordered as one of the m prefix operations belonging to group $\lceil \frac{j}{m} \rceil$.

If a SUGGEST operation S performs the F&I on line s10, receiving $x < k$, then S will be ordered as the suffix operation belonging to group $k - x$.

▶ **Lemma 14.** *When a* SUGGEST *operation completes iteration j of the for loop on line s2, position$[j]$ is filled.*

Proof. Let S be a SUGGEST operation by process p_i that completes iteration j. If, during SUGGEST operation S, p_i reads $A_j[i] = \bot$, then it performs the COMPETE operation on *position*$[j]$ in line s5 before iteration j completes. If, on the other hand, p_i reads $A_j[i] \neq \bot$ on line s3, then there must have been a SUGGEST operation S' by process p_i that completed before p_i began operation S, in which the COMPETE operation on *position*$[j]$ in line s5 was executed. In either case, by the time that S completes iteration j, at least one COMPETE operation has been performed on *position*$[j]$. The first of these COMPETE operations must have returned *true*, so *position*$[j]$ is filled. ◀

From Lemma 14, we get the following important corollaries:

▶ **Corollary 15.** *All* SUGGEST *operations that perform line s10 can be ordered after all* SUGGEST *operations that fill some position.*

Proof. A SUGGEST operation S that performs line s10 must first perform all km iterations of the for loop on line s2. It follows from Lemma 14 that there is no SUGGEST operation that fills a position and starts after S completes. ◀

▶ **Corollary 16.** *For $j < j'$, a* SUGGEST *operation that fills position$[j]$ can be ordered before a* SUGGEST *operation that fills position$[j']$.*

Proof. Consider a SUGGEST operation S that fills *position*$[j]$ and a SUGGEST operation S' that fills *position*$[j']$, with $j < j'$. To fill *position*$[j']$, S' must have first completed iteration j of the for loop in line s2. Thus, by Lemma 14, *position*$[j]$ was already filled before S' completed. By assumption, *position*$[j]$ was filled by operation S. Hence, S can be linearized before S'. ◀

▶ **Observation 17.** *If the* F&I *operation performed on line s10 by* SUGGEST *operation S returns x, and the* F&I *operation performed on line s10 by* SUGGEST *operation S' returns $x' > x$, then S can be ordered before S'.*

We are now able to order the SUGGEST operations in execution $\mathcal{E}$. In particular, by Corollary 16, the SUGGEST operations that fill *position*$[\ell]$, for $(j-1)m + 1 \leq \ell \leq jm$, can be ordered as the m prefix operations of group j. Moreover, by Corollary 15 and Observation 17, the SUGGEST operation that receives $x < k$ from the F&I on line s10 can be ordered as the suffix operation of group $k - x$ and any SUGGEST operation that receives $x > k - 1$ from the F&I on line s10 can be ordered after the first $km + k - 1$ operations.

All that remains is to order the prefix operations belonging to each group with respect to each other and show that the results of the operations in this ordering are consistent with the specifications of $O_{m,k}$. We first make the following two observations:

▶ **Observation 18.** *If any* SUGGEST *operation that belongs to group g returns a value, it returns the decision of cons$[g]$.*

▶ **Observation 19.** *If the suffix operation belonging to group g proposes to cons$[g]$, it proposes the value of the* SUGGEST *operation that fills position$[gm]$.*

In execution $\mathcal{E}$, if no SUGGEST operation belonging to group g proposed to *cons*$[g]$, then, by Corollary 16, we can order the prefix operations in order of the indices of the Q_1 objects they fill.

Otherwise, one or more values were proposed to *cons*$[g]$. By Observation 19, only the arguments of the prefix operations belonging to group g are proposed to *cons*$[g]$. Let v be the value first proposed to (and, hence, decided by) *cons*$[g]$ and let j be the smallest index, with $(g-1)m+1 \leq j \leq gm$, such that the prefix operation S that fills *position*$[j]$ has argument v. No prefix operation in group g that fills *position*$[j']$, for $j' < j$, finished before S began, since v is the value first proposed to *cons*$[g]$. It follows that we can order every other prefix operation belonging to group g after S, in order of the indices of the Q_1 objects they fill.

By Observations 18 and 19, the results of the SUGGEST operations in this ordering are consistent with the sequential specifications of the $O_{m,k}$ object. Note that the implementation of SUGGEST in Algorithm 3 is wait-free. Thus, we have shown the following theorem:

▶ **Theorem 20.** *There is a wait-free implementation of the $O_{m,k}$ object from $(m+1)$-consensus objects and registers for any $m \geq 2$ and any $k \geq 2$.*

5 Conclusions

In this paper, we introduced the Q_r object for $r \geq 0$, showed that it has consensus number $r+2$, and presented a wait-free implementation of Q_r from $(r+2)$-consensus objects and registers in a system with any finite number of processes. Using this object, we showed that, for any $m, k \geq 2$, there is a wait-free implementation of $O_{m,k}$ from $(m+1)$-consensus objects and registers in a system with any finite number of processes. This means that there is an infinite sequence $O_{m,2}, O_{m,3}, \ldots$ of objects with increasing computational power, all of which have consensus number m and are computationally less powerful than the $(m+1)$-consensus object. From Rachman's result, we know that for every $m' > m \geq 1$, there exists a nondeterministic object X with consensus number m which cannot be implemented using m'-consensus objects and registers in some system with a finite number of processes. If the same property is true when restricted to deterministic objects, our implementation of $O_{m,k}$ from $(m+1)$-consensus objects and registers shows that $O_{m,k}$ objects cannot be used to prove this.

The $O_{m,k}$ objects, for $k \geq 2$, are the only known deterministic objects of consensus number m that cannot be implemented in a wait-free manner by m-consensus objects and registers for some finite number of processes. It may be the case that any deterministic object with consensus number m has a wait-free implementation from $(m+1)$-consensus objects in a system with any finite number of processes. One approach to proving this is to show that, for every object O with consensus number m, there exists $k \geq 2$ such that O is computationally less powerful than $O_{m,k}$. Another interesting question is whether there exists some deterministic object with consensus number m that can implement any other deterministic object with consensus number m in a system with any finite number of processes.

Acknowledgements. I would like to thank my supervisor, Faith Ellen, for her constant encouragement and invaluable guidance and my anonymous reviewers for their very helpful feedback. The implementation of $(r+2)$-consensus from Q_r objects and registers given in Algorithm 1 is due to Leqi Zhu, who simplified my original implementation. This work has been generously supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

1 Yehuda Afek, Faith Ellen, and Eli Gafni. Deterministic objects: Life beyond consensus. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 97–106, 2016.

2 Yehuda Afek, Eli Gafni, and Adam Morrison. Common2 extended to stacks and unbounded concurrency. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Principles of Distributed Computing*, pages 218–227, 2006.

3 Yehuda Afek, Eli Gafni, John Tromp, and Paul M. B. Vitányi. Wait-free test-and-set (extended abstract). In *Proceedings of the 6th International Workshop on Distributed Algorithms (WDAG)*, pages 85–94, 1992.

4 Yehuda Afek, Eytan Weisberger, and Hanan Weisman. A completeness theorem for a class of synchronization objects. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 159–170, 1993.

5 Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

6 Ophir Rachman. Anomalies in the wait-free hierarchy. In *Proceedings of the 8th International Workshop on Distributed Algorithms (WDAG)*, pages 156–163, 1994.