On Consensus Number 1 Objects

Pankaj Khanchandani
Adobe Systems
Bangalore, India
kpankaj@adobe.com

Jan Schäppi
ETH Zurich
Zurich, Switzerland
schajan@ethz.ch

Ye Wang
ETH Zurich
Zurich, Switzerland
wangye@ethz.ch

Roger Wattenhofer
ETH Zurich
Zurich, Switzerland
wattenhofer@ethz.ch

***Abstract*—The consensus number concept is used to determine the power of synchronization primitives in distributed systems. Recent work in the blockchain domain motivates shifting the attention to consensus number 1 objects, as it has been shown that transaction-based blockchains just need consensus number 1. In this paper we want to get a better understanding of such consensus number 1 objects.**

In particular, we study the necessary and sufficient conditions for determining the consensus number 1 objects. If an object has consensus number 1, then its operations must be either commutative or associative (necessary condition). On the other hand, if the operations are *consistently* commutative or overwriting, i.e., independent of the current state of the object, then the consensus number of the object is 1 (sufficient condition). We give an algorithm to implement such generic consensus number 1 objects using only read/write registers. This implies that read/write registers are universal enough to solve tasks, such as asset transfer of a cryptocurrency, among many others, in wait-free distributed systems for any number of processes.

***Index Term*— Consensus number, synchronization hierarchy, object implementation, read/write registers**

I. Introduction

In distributed systems, we typically have multiple processes concurrently working on the same data. Distributed systems provide tools to orchestrate synchronization, both on the hardware and the software level. Using these synchronization primitives, one process will not inadvertently overwrite the data of another concurrent process. We can measure the power of a synchronization primitive in asynchronous distributed systems by the so-called consensus number introduced by Maurice Herlihy [7].

The consensus number of a synchronization primitive (often also called: object) determines how many processes can achieve consensus. More precisely, an object has consensus number n, if n processes can achieve consensus, but $n+1$ processes cannot. These n processes then can implement any wait-free object that is shared among n processes using the consensus number n object and read/write registers [7], [8], [14].

Back when the consensus number was introduced, designers of hardware and concurrent programming languages were motivated to provide synchronization primitives with a high consensus number. Having consensus allows for virtually any computation, since in the worst case, the n concurrent processes can get consensus for just about anything before they move on with the next task. Not surprisingly, all future hardware and software provided synchronization primitives which allowed an unbounded consensus number, i.e., $n \to \infty$.

However, a high consensus number is often also associated with a high implementation cost and low parallelism, which is a disadvantage. Recently, several works started advocating low consensus number primitives instead. In the important application domain of blockchains for instance, multiple teams started questioning the importance of consensus at all, e.g. [6], [18].

It turns out that simple blockchains for financial transactions (equivalent to what Bitcoin [13] offers) do not need consensus at all, and so these transactions can be implemented with a synchronization mechanism that does not need a high consensus number. Of particular interest are hereby primitives that merely have consensus number 1, and hence cannot even achieve consensus among 2 concurrent processes.

Indeed, if we only do financial transactions, why would consensus be needed? If a person A wants to send money to another person B, A can simply sign a transaction. All blockchain participants will be informed about this transaction via a reliable broadcast protocol [4]. Consensus is not needed unless person A tries to send the same money not only to B but also (concurrently) to C. This is known as a double spend attack. In the case of such a double spend, the blockchain would need consensus to decide whether the money is transferred to B or C. In consensus-free blockchains (e.g., [3], [6], [18]), the money would be sent to either B or C, or, A could even lose the money. In transaction-based blockchains, losing the money seems plausible, since the the problem only occurred because A tried to cheat. On the other hand, consensus-free blockchains have many advantages over consensus-based blockchains: They are simpler, more efficient, and also work in completely asynchronous environments.

But the use of consensus number 1 goes well beyond crypto transactions. For instance, snapshot [1] is another typical example of a useful consensus number 1 object. In this paper, we want to put a spotlight on consensus number 1 objects. We want to understand their possibilities and limitations better.

We first consider how to determine consensus number 1 objects depending on the relationships between operations. We prove that the operations of a consensus number 1 object must satisfy either commutative or overwriting relationships with each other. Otherwise, we can construct an algorithm with the object and read/write registers to achieve binary agreement between the two processes. Then, we study the implementation of consensus number 1 objects with read/write registers. We propose an algorithm to simulate consensus number 1 objects whose operations keep the same commutative and overwriting relationships at every system state. Meanwhile, this result also implies that if the relationship between two operations is consistent across the system states, then the object has consensus number 1.

II. Related Work

Since Herlihy [7] introduced the hierarchy based on the consensus number of wait-free objects, it has been used to quantify the synchronization power of the objects [15], [16]. He analyzed the consensus number of different objects: those with a finite consensus number, such as fetch-and-add, and those with an infinite consensus number, such as compare-and-swap. On top of determining the consensus number of objects, Herlihy also considered universality. He showed that an object of consensus number n could implement any other objects in a system of no more than n processes. Nevertheless, it is not clear how to implement other objects with objects of consensus numbers lower than n. In particular, Herlihy proposed an open question regarding the wait-free hierarchy: can read/write registers implement any object with consensus number 1 in a system of two or more processes?

Later, Jayanti studied how to utilize objects of consensus numbers lower than n to implement other objects. Although it has been proved that it is impossible to implement an object of a higher consensus number with an object of a lower consensus number [7], [17], Jayanti showed that this wait-free hierarchy is not robust if combining multiple objects with lower consensus [10]. In particular, he proposed a type weak-sticky object of consensus number k, which can implement an object of consensus number $k+1$ together with read/write registers. However, their results are not generalized to the most basic objects with consensus number 1.

Recently, there is a series of works analyzing the computational power of low consensus objects [5], [11], [12]. However, they do not consider objects as individual elements in the systems. They model a set of registers that support the set of synchronization operations. For example, in traditional systems, a consensus number 1 object only supports either the decrement operation or the multiply operation; while in their models, an object can execute both decrement operation and multiply operation atomically. Ellen et al. showed that with two consensus number 1 operations, such as decrement and multiply, the system could achieve the binary agreement for any given number of processes. Khanchandani et al. [11] implemented the concurrent queue objects with the compare-and-swap operation and other low consensus number operations. Later, Khanchandani et al. [12] utilized another two elementary operations, i.e., half-max and max-write [2], to realize a wait-free and linearizable implementation of the compare-and-swap object, which further decreases the consensus number of operations in the system. However, the objects that are studied in these works are not consensus number 1 anymore. Although they only execute low consensus number operations, the inherent atomicity between different operations increases the synchronization power of these objects.

The implementations of consensus-less objects in cryptocurrency system [6], [18] motivate us to revisit the computational power of consensus number 1 objects. Guerraoui et al. [6] show that even if all processes have not achieved the consensus in the distributed system, it is still possible to ensure the correctness of a cryptocurrency system. Furthermore, the consensus number of a cryptocurrency object can be as low as 1 if each account is only operated by a single process. These findings suggest that objects with a finite consensus number may also be universal to solve other tasks in asynchronous wait-free distributed systems with any number of processes [7], which motivates us to study the computational power of these objects that are usually overlooked.

In this paper, we aim to answer the open question [7]: can read/write registers implement any object with consensus number 1 in a system of two or more processes? To the best of our knowledge, this paper is the first to explore the implementation of an arbitrary object with read/write registers for unbounded number of processes. We find that if every pair of operation of the objects consistently follows either the commutative or the overwriting relationship, the consensus number 1 objects can be implemented by read/write registers.

III. Preliminaries

In this section, we define some frequently used terms.

An object O is modeled by an I/O automaton: $\langle S, \Sigma, \delta, s_0, F\rangle$. $S = \{s_j\}$ is the state set, where $j \in \Gamma_Q$. $\Sigma = \{op_k\}$ is the operation set, where $k \in \Gamma_\Sigma$. Both Γ_Q and Γ_Σ are (infinite) index sets. $\delta : Q \times \Sigma$ is the transition function, while each operation op_j causes a state transition on any state s_i, i.e., $s'_i = s_i + op_j$, and returns a response. $s_0 \in S$ is the initial state and $F \subseteq S$ is the set of terminal state. There also exists a $read_O$ operation that returns the current state of O.

We define two relationships between operations of O.

Definition 1 (Commutative Operations)**.** *Consider a state s and two operations op_1 and op_2 of an object O. If the following equation holds:*

$$s + op_1 + op_2 = s + op_2 + op_1,$$

then we define op_1 and op_2 are commutative at state s. If for any $s \in S$, op_1 and op_2 are commutative, then op_1 and op_2 are commutative operations in O.

Definition 2 (Overwriting Operations)**.** *Consider a state s and two operations op_1 and op_2 of an object O. If the following equation holds:*

$$s + op_1 = s + op_2 + op_1,$$

then we define the relationship between op_1 and op_2 as op_1 overwriting op_2 at state s. If for any $s \in S$, op_1 overwrites op_2 at s, then op_1 overwrites op_2 in O.

The execution of operations will occur events in the system, while these events make up execution histories. We further define these terms as follows.

Definition 3 (Events)**.** *Let op be an operation. The execution of op by a process p is modeled by two events: an invocation event, denoted as $invoc(op)$, which occurs when process p invokes op, and a response event, denoted as $resp(op)$, which occurs when p terminates the operation.*

Definition 4 (Histories)**.** *A $history$ is a total order on the events produced by processes. Given any two events e_1 and e_2, $e_1 < e_2$ if e_1 happens before e_2 in the corresponding history. We always have either $e_1 < e_2$ or $e_2 < e_1$. A history is denoted as $\hat{H} = \langle E, < \rangle$, where E is the set of events.*

Definition 5 (Linearizable history)**.** *A history $\hat{H} = \langle E, < \rangle$ is linearizable if there is an equivalent sequential history $\hat{H_{seq}} = \langle E, <_{seq} \rangle$ where the sequence of effective operations issued by processes satisfies that each effective operation appears as executed at a single point of the time line between its invocation event and its response event.*

Definition 6 (Concurrent operations)**.** *Given a history $\hat{H} = \langle E, < \rangle$ and two effective operations op_1 and op_2. We say op_1 precedes op_2 if $invoc(op_2) > resp(op_1)$. If neither $invoc(op_1) > resp(op_2)$ nor $invoc(op_2) > resp(op_1)$, then we say that op_1 and op_2 are concurrent.*

Definition 7 (Internal events)**.** *Given a history $\hat{H} = \langle E, < \rangle$ of an object O implemented by Algorithm 4 and Algorithm 5.*

- *For an operation $op(arg)$, there are two internal events: a scan event, denoted as $scan(op)$, which occurs at the linearization point of the scan operation of the atomic snapshot object, and an update event, denoted as $upd(op)$, which occurs at the linearization point of the update operation of the atomic snapshot object.*
- *For a read operation $read()$, there is one internal event: a scan event, denoted as $snap(op)$, which occurs at the linearization point of the snapshot operation of the atomic snapshot object.*

Definition 8 (Effective concurrent operations)**.** *Given a history $\hat{H} = \langle E, < \rangle$ of an object O implemented by Algorithm 4 and Algorithm 5, and two operations op_1 and op_2. We say op_1 precedes op_2 if $scan(op_2) > upd(op_1)$. If neither $scan(op_1) > upd(op_2)$ nor $scan(op_2) > upd(op_1)$, then we say that op_1 and op_2 are effectively concurrent.*

In this paper, we study objects which are atomic. An atomic object O satisfies the following three properties,

- deterministic: for each state s_i and operation op_j, the state transition and the response only depends on s_i and op_j;
- oblivious: every process p can invoke every operation; moreover, the state transition and the response does not depend on p;
- atomic (linearizable): the execution of O can be specified by linearizable history [9].

IV. Determining Consensus Number 1 Objects

In this section, we study consider the most central question: which objects are consensus number 1? We find that given a particular state s_i, the relationship between any two operations of a consensus number 1 object must be either overwriting or commutative. Otherwise, we can always find an algorithm that leads to agreements between two processes. In other words, the overwriting and commutative relationships of operations are necessary conditions of consensus number 1 objects.

Theorem 1. *If an atomic object $O = \langle S, \Sigma, \delta, s_0, F \rangle$ does not satisfy the following property, then the consensus number of O is at least 2.*

For each state $s_i \in S$ and operations $op_j, op_k \in \Sigma$, one of the following equations must hold,

- $s_i + op_j + op_k = s_i + op_k + op_j$
- $s_i + op_j + op_k = s_i + op_k$
- $s_i + op_j = s_i + op_k + op_j$

Proof. We prove Theorem 1 by contradiction. We show that if the statement of Theorem 1 does not hold, then there exists an algorithm that enables two processes P and Q achieve agreement with an object O and two read/write registers.

Assume that there exists a state s_i and two operations op_j and op_k of a consensus number 1 object O, such that the following three inequalities hold at the same time. In other words, op_j and op_k are neither commutative nor overwriting with s_i.

- $s_i + op_j + op_k \neq s_i + op_k + op_j$
- $s_i + op_j + op_k \neq s_i + op_k$
- $s_i + op_j \neq s_i + op_k + op_j$

Let us denote $s_j = s_i + op_j$, $s_k = s_i + op_k$, $s_{jk} = s_i + op_j + op_k$, and $s_{kj} = s_i + op_k + op_j$. From the assumption, we know that $s_j \neq s_{kj}$, $s_{jk} \neq s_{kj}$, and $s_k \neq s_{jk}$.

Let P and Q be two processes that share a two-register array $prefer$, where each entry is initialized to $\perp$, and an object O, initialized to s_i. Process P and Q execute the protocols shown in Algorithm 1.

Note that, P may observe three possible states of O in $decide_P$: s_j, s_{jk}, and s_{kj}. All of these three states are different, so p can easily distinguish them and get the information whether op_j or op_k executes first. If op_j executes first, the protocol chooses the input of P, otherwise it chooses the input of process Q. If op_k executes first, then $prefer[Q] \neq \perp$. The return of $decide_P$ must be $value_Q$. With the same argument, Q will return $value_Q$ if op_k executes first and $value_P$ if op_j executes first.

Therefore, two processes achieve agreement by Algorithm 1 with an object O and two read/write registers, which turns out that the object O has consensus number at least 2, which contradicts our assumption.

□

V. Implementing Consensus Number 1 Objects

In section IV, we show that if two operations are not commutative or overwriting at any system states, then the object

Algorithm 1 Code for process P and Q, two-processes agreement

```
 1: function $decide_P(value_P)$
 2:     $prefer[P] \leftarrow value_P$
 3:     execute $op_j$ on $O$
 4:     $CurState \leftarrow read_O()$
 5:     if $CurState = s_j$ or $CurState = s_{jk}$ then
 6:         return $prefer[P]$
 7:     else
 8:         return $prefer[Q]$
 9:     end if
10: end function
 1: function $decide_Q(value_Q)$
 2:     $prefer[Q] \leftarrow value_Q$
 3:     execute $op_k$ on $O$
 4:     $CurState \leftarrow read_O()$
 5:     if $CurState = s_k$ or $CurState = s_{kj}$ then
 6:         return $prefer[Q]$
 7:     else
 8:         return $prefer[P]$
 9:     end if
10: end function
```

has consensus at least number 2. The remaining question is: are all objects satisfy Theorem 1 consensus number 1 objects?

In this section, we answer this question by providing an implementation algorithm of consensus number 1 objects with read/write registers. Read/write registers are the most elementary objects in distributed systems with consensus number 1. If an object can be implemented with read/write registers, then we can ensure its consensus number as 1.

We show that if the relationship of two operations is consistent at any state, then the object has consensus number 1.

Theorem 2 (consensus number 1 objects)**.** *If an atomic object $O = \langle S, \Sigma, \delta, s_0, F \rangle$ satisfies the following property, then it has consensus number 1.*

For any operations $op_j, op_k \in \Sigma$, one of the following equations must always hold with all $s_i \in S$,

- $s_i + op_j + op_k = s_i + op_k + op_j$
- $s_i + op_j + op_k = s_i + op_k$
- $s_i + op_j = s_i + op_k + op_j$

To prove Theorem 2, we provide an algorithm to implement objects that satisfy the statement with only read/write registers. We first study the properties that these objects have and consider how to construct the algorithm protocols based on these properties.

A. Overwriting Graph of Objects

If all operations are commutative with each other, then it is easy to construct an algorithm to implement the object O with the read/write registers because the order of operations does not influence the final state. However, because of the overwriting relationship between operations, we should carefully consider the order of executing operations.

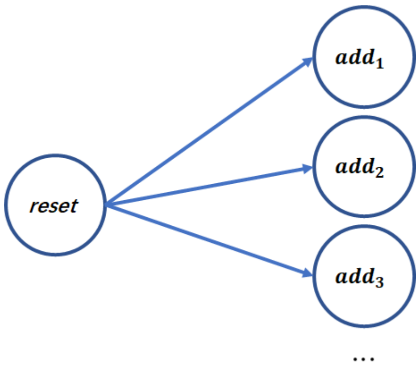

Fig. 1. The Overwriting graph of the add/reset object. The $reset$ operation overwrites all other add operations, so there are edges pointing from $reset$ to other processes.

To get a more clear idea of the relationship between operations, we construct an overwriting graph $G_O = (\Gamma_\Sigma, E)$. Each process v in G represents an operation $op_v \in \Sigma$, while if op_v overwrites op_w, then there is a directed edge $(v, w) \in E$ pointing from v to w.

Consider an example of the add/reset object. The state of the object is an integer, which is initialized to 0. An add_x operation adds value x to the current state and a $reset$ operation make the state to 0. The $reset$ operation overwrites all other add operations and the Overwriting graph of the add/reset object is shown in Figure 1.

Moreover, the overwriting graph has some special properties given the Overwriting and commutative relationships between operations.

Lemma 1. *If there is a path in $G_O = (\Gamma_\Sigma, E)$ from process v to process w, then there exists an edge directly pointing from process v to process w.*

Proof. We prove Lemma 1 by induction of the path length n.

When $n = 1$, then the edge directly points from v to w.

When $n > 1$ and the lemma holds for all paths with length $n-1$. Let us consider a path of length $n : v \to 1 \to 2 \to \ldots \to n-1 \to w$, then there is a direct edge pointing from v to $n-1$. We know that for a state s, the following equations always holds,

Now we consider, if the lemma holds for all paths with length $n-1$, then there is a path from v to w of length 2, i.e. $v \to u \to w$. For any state s, the following equations always holds,

$$s + op_w + op_{n-1} + op_v = s + op_w + op_v,$$

and

$$s + op_w + op_{n-1} + op_v = s + op_{n-1} + op_v = s + op_v.$$

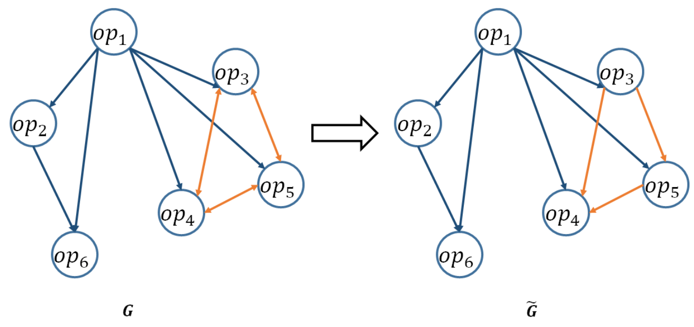

Fig. 2. Construct a contracted graph $\tilde{G}$ from the overwriting graph G. processes op_3, op_4, op_5 form a clique in G. We randomly order them as $op_3 > op_5 > op_4$ and deleted edges.

Therefore, for any state s, $s + op_w + op_v = s + op_v$, which denotes op_v overwrites op_w and there is an edge directly pointing from process v to process w in G_O. □

Note that if two operations are commutative, then there is no edge and no path between two corresponding processes in $G_O = (\Gamma_\Sigma, E)$.

Lemma 2. *If processes v and w are in the same clique of graph $G_O = (\Gamma_\Sigma, E)$, then the indegree and the outdegree of v and w are the same, i.e., $deg^+(v) = deg^+(w), deg^-(v) = deg^-(w)$.*

Proof. We prove Lemma 2 by contradiction.

Assume there exists a process u pointing to process v and there is no edge from u to w. Because v and w are in the same clique, there is a path of length 2 from u to w, i.e., $u \to v \to w$. According Lemma 1, there is also an edge from u to w, which contradicts to our assumption.

The same argument also works for outdegree. Assume there exists an edge from v to u and no edge from w to u. Because v and w are in the same clique, there is a path of length 2 from w to u, i.e., $w \to v \to u$. According Lemma 1, there is also an edge from w to u, which contradicts to our assumption. □

B. Contracted Overwriting Graph

For each overwriting graph, we can construct a contracted graph of the object O.

Definition 9. *The contracted overwriting graph $\tilde{G} = (\Gamma_\Sigma, \tilde{E})$ is a directed graph derived from an overwriting graph G. For each clique, we randomly assign a priority order among all operations and delete edges from the low priority processes to the high priority processes.*

We present an example of constructing contracted overwriting graph in Figure 2. processes op_3, op_4, op_5 form a clique in G. We randomly order them as $op_3 > op_5 > op_4$ and remove edges $(op_4, op_3), (op_4, op_5), (op_5, op_3)$ in $\tilde{G}$.

Lemma 3. *There is no cycle in a contracted overwriting graph $\tilde{G}$.*

Proof. We prove Lemma 3 by contradiction.

Assume there is a cycle in a contracted overwriting graph $\tilde{G}$: $(u_1, u_2), (u_2, u_3), .., (u_n, u_1)$. By Lemma 1, we know that there are also edges between any process pairs in this cycle. Then, processes $(u_1, u_2, u_3, \dots, u_n)$ form a clique. However, there is no clique in a contracted overwriting graph, which contradicts to our assumption. □

With the same argument in Lemma 1, we can have the following lemma in contracted overwriting graphs as in overwriting graphs.

Lemma 4. *If there is a path in $\tilde{G}$ from process v to process w, then there exists an edge directly pointing from process v to process w. Moreover, every operation in OP_v overwrites every operation in OP_w.*

Since $\tilde{G}$ has no cycles, we can assign levels to every operation using Algorithm 2. Trivially operations with the same levels commute with each other.

Algorithm 2 Determining the levels for all operations

```
 1: function OPERATIONLEVEL
 2:     $\tilde{G} = (\Gamma_\Sigma, \tilde{E})$ is the contracted overriding graph of $O$
 3:     $lev \leftarrow$ the length of the longest path in $\tilde{G}$
 4:     while $|\Gamma_\Sigma| > 0$ do
 5:         $U_{Top} \leftarrow \{u \in \Gamma_\Sigma : in(u) = \emptyset\}$
 6:         $\Gamma_\Sigma \leftarrow \Gamma_\Sigma \setminus U_{Top}$
 7:         $\tilde{E} \leftarrow \tilde{E} \setminus out(U_{Top})$
 8:         for $u \in U_{Top}$ do
 9:             $level[\{op_k | k \in u\}] = lev$
10:         end for
11:         $lev \leftarrow lev - 1$
12:     end while
13: end function
```

C. Consensus Number 1 Object Algorithm

We prove Theorem 2 by proposing algorithms (Algorithm 4, Algorithm 5) to implement consensus number 1 object O with read/write registers. We use a shared atomic snapshot object to realize such algorithm [1]. Note that snapshot objects can be implemented by read/write registers and are consensus number 1 [1].

Algorithm 3 State of processes

```
1: state
2:     $snapshot$          ▷ A shared atomic snapshot object,
   process $p$ can modify the $p-th$ element
3:     $level$   ▷ A shared register vector. $level[op_k]$ denotes
   the level of $op_k$
4:     $MaxLevel$    ▷ The length of the longest path in $\tilde{G}$,
   the highest level of operations
5:     $P$                                    ▷ Set of processes
6:     $reg$        ▷ Each process $p$ has a local variable $reg_p$,
   keeping historical executed operations
7: end state
```

The basic idea is as follows: All operations applied to the object are recorded in an atomic snapshot, which is shared by all processes. When a process p wants to execute an operation op_i to the object, it first takes a scan of the snapshot to know which operations have been recorded in the snapshot advance to it. It will consider these operations have been executed in the system. Then, it adds op_i and the observations to its operation list and updates to the snapshot object. When process p executes a read operation, it first scans operations which are recorded in the snapshot object and then finds a justifiable order to apply to the initial state.

Algorithm 4 Executing an operation, process p

```
1: function $op_i(args)$
2:     $mem \leftarrow snapshot.scan()$ ▷ read operations executed before $op_i$
3:     $op.com = op_i(args)$
4:     $op.mem = mem$
5:     $reg_p \leftarrow reg_p + op$    ▷ add $op_i$ to the operation list
6:     $snapshot.update(reg_p)$ ▷ update the operation list of process $p$
7: end function
```

Algorithm 5 Read Operation, process p

```
1: function $read$
2:     $mem \leftarrow snapshot.scan()$ ▷ read operations has been executed
3:     $OP \leftarrow \emptyset$    ▷ set of executed operations
4:     for $q \in P$ do
5:         $OP \cup mem[q]$
6:     end for
7:     $s \leftarrow s_0$    ▷ start with the initial state of $O$
8:     $S \leftarrow \emptyset$
9:     for $u \in \Gamma_\Sigma$ do
10:        $pred(op) = \{\tilde{op} | \exists$ path from $\tilde{op}$ to $op$ in $\tilde{G}\}$ ▷ operations which overwrite $op$
11:    end for
12:    $lev \leftarrow MaxLevel$
13:    for $lev \geq 0$ do
14:        $S_+ \leftarrow \{op \in OP | level(op.com) = lev, \forall \tilde{op} \in S \cap pred(op), \tilde{op} \in op.mem\}$
15:            ▷ operation which have not been overwritten by other operations
16:        $OP \leftarrow OP \setminus S_+$
17:        $S \leftarrow S \cup S_+$
18:        $lev \leftarrow lev - 1$
19:        for $op \in S_+$ do
20:            $s = s + op.com$    ▷ operations in $S_+$ are commutative
21:        end for
22:    end for
23:    return $s$
24: end function
```

Note that, as some operations overwrite others, some operations will not actually have an influence on the state. The algorithm categorizes operations into different groups, while some of them will not influence the final state (remains in the OP set by the end of the algorithm) and the rest of them should be applied to the object (operations in the S_+ set).

We start with operations with highest level (Algorithm 5 Line 12). Note that these operations will never be overwritten by other operations because they have no incoming degrees (Algorithm 2). Moreover, these operations are commutative. Thus, the executed order does not influence the final state of these operations (Algorithm 5 Line 20).

In the substantial iterations, we consider operations of lower levels. Note that if the operation op with lower levels contributes to the final state of the object, there are no operations that can overwrite it (Line 10) have been executed after it. In other words, if $\tilde{op} \in op.mem$, then the execution time of $snapshot.scan()$ in op is after the execution time of $snapshot.update()$ in $\tilde{op}$. Because we cannot identify the invoke time and the response time of both op and $\tilde{op}$, while the execution time of $snapshot.scan()$ must be after the invoke time and the execution time of $snapshot.update()$ must be before the response time, we infer that op is executed after $\tilde{op}$. Therefore, op is not overwritten by $\tilde{op}$ and may influence the final state of the object.

If an operation op is selected in line 14, then the relationship between op and operations $\tilde{op}$ selected in previous iterations has two possibilities. First, op can be overwritten by $\tilde{op}$ but it executes after $\tilde{op}$. In such case, there is no argument to apply op after $\tilde{op}$ (line 20). Second, op and $\tilde{op}$ are commutative. Thus, there is no difference between different execution orders to the final state. Note that it is impossible that op overwrites $\tilde{op}$. Otherwise, $\tilde{op}$ will have a lower level than op and will not be selected in previous iterations.

D. Correctness of Consensus Number 1 Object Algorithm

We show the correctness of consensus number 1 object algorithm by proving it is deterministic, oblivious, and atomic. We first argue the algorithm satisfy the first two properties of the atomic object and then prove its atomicity.

Lemma 5. *An object O implemented by Algorithm 4 and Algorithm 5 is deterministic and oblivious.*

It is trivial that an object O implemented by Algorithm 4 and Algorithm 5 is deterministic and oblivious. We have not accessed to any random source for processes determining their actions and each process share the same algorithm protocol. To prove Theorem 2, we show that O is also atomic in the next lemma.

Lemma 6. *Given a history $\hat{H} = \langle E, < \rangle$ of an object O implemented by Algorithm 4 and Algorithm 5, we can always find an equivalent sequential history $\hat{H_{seq}} = \langle E, <_{seq} \rangle$ where the sequence of operations issued by processes satisfies that each operation appears as executed at a single point of the time line between its invocation event and its response event.*

Proof. We present a way to construct $\hat{H_{seq}} = \langle E, <_{seq} \rangle$ and show that it satisfies the atomicity.

For each $read$ operation, the linearization point in $\hat{H_{seq}}$ is equal to the linearization point of the $scan$ operation at $\hat{H_{seq}}$, i.e., $lin(read) = scan(read)$.

Then we define the linearzation point of other operations. We start from the operations observed by the first $read$ operation. If the operation op has appeared in the set S by the end of Algorithm 5, then we determine the linearzation point of op as the linearzation point of its update operation, i.e., $lin(op) = upd(op)$.

Otherwise, there exists an operation $\tilde{op} \in S$ such that $\tilde{op}$ overwrites op. If $upd(op) \leq upd(\tilde{op})$, then we set the linearzation point of op as the linearzation point of its update operation, i.e., $lin(op) = upd(op)$. Otherwise, we set the linearzation point of op before the linearzation point of $\tilde{op}$, i.e., $lin(op) = lin(op) - \epsilon$.

Note that $\tilde{op} \notin op.mem$, if $upd(op) > upd(\tilde{op})$, then op and $\tilde{op}$ are effective concurrent, and $invoc(op) < scan(op) < upd(\tilde{op}) < upd(op) < resp(op)$. Therefore, the linearization point of op is valid because $invoc(op) < lin(op) = lin(\tilde{op}) - \epsilon < resp(op)$.

Then we show that the linearization of operations in $\hat{H_{seq}}$ is equivalent to $\hat{H}$ until the first $read$ operation.

Let us denote the sequence of the operation in $\hat{H_{seq}}$ as $op_1, op_2, \ldots, op_k$. In the first step, we remove all operations which are not in S by the end of the first $read$ operation. We first consider an operation op_j, which is not in S with the lowest level. There must exist another operation op_k which overwrites op_j where $j < k$. We consider op_j and op_{j+1}. Because op_j has the lowest level, it will not overwrite any other operations. It will be overwritten by op_{j+1}, or they are commutative. In the first scenario, we can directly remove op_j from the sequence and do not influence the final state. In the second scenario, we can exchange the position of op_j, and op_{j+1} and do not influence the final state. We repeat this exchange until the $k-th$ position, and then op_j eventually is removed from the sequence because op_k overwrites op_j. The sequence without op_j results in the same state as $\hat{H_{seq}}$. We recursively apply this deduction until all remaining operations are in S.

In the second step, we reorder the operations in the sequence. We start with the operation op_l, which is in S with the highest level. We exchange op_l with op_{l-1} until there is no other operation with lower level in front of it in the sequence. Because op_l has higher level than op_{l-1} and op_{l-1} cannot be overwritten by op_l, this exchange always works. Otherwise, op_{l-1} will not exist in S by the end of the first $read$ operation. Because $lin(op_{l-1}) < lin(op_l)$, $op_l \notin op_{l-1}.mem$. If op_l overwrites op_{l-1}, then op_{l-1} will not be selected in line 14. Thus, we can exchange op_{l-1} and op_{l-1} and do not influence the object state.

After reordering the operations, we obtain the same sequence order as operations have been applied to the object in the first $read$ operation, which proves that until the first $read$ operation, $\hat{H_{seq}}$ is equivalent to $\hat{H}$.

Then we consider operations which have been observed by the second $read$. We follow the same rules as for operations observed by the first $read$: for operation op that appears in S by the end of the second $read$, $lin(op) = upd(op)$; for operation op that is not in S, compare its update time with the Overwriting operation $\tilde{op}$, such that $lin(op) = \min(upd(op), lin(\tilde{op})\epsilon)$. Note that $lin(op)$ might be smaller than the linearization time of the first $read$ operation. However, it is only possible if $lin(op) = lin(\tilde{op})\epsilon$, otherwise, the first $read$ operation can observe op. op does not contribute to the object state and it immediately overwrite by $\tilde{op}$. Therefore, it does not influence the equivalence between $\hat{H_{seq}}$ and $\hat{H}$ for the first $read$ operation.

Then we use the same argument to show that $\hat{H_{seq}}$ is equivalent to $\hat{H}$ until the second $read$ operation and vice versa. For operations which are not observe by the last $read$ operation, we denote their linearization time as its update time, i.e., $lin(op) = upd(op)$.

Until now, we have defined the linearization point of all operations in $\hat{H_{seq}}$, which is equivalent to $\hat{H}$. We prove that every history corresponding to an execution of O is linearizable.

□

VI. Conclusion

In this work, we investigate the consensus number 1 objects in distributed systems. We contribute to the understanding of the synchronization hierarchy structure of consensus numbers.

Compared to previous work, we are the first to study the implementation of consensus number 1 objects for unbounded number of processes using only read/write registers and provide an algorithm to implement a class of consensus number 1 objects.

We show that the commutative relationship and the overwriting relationship between operations are necessary for consensus number 1 objects. Moreover, the *consistent* commutative and overwriting relationships between operation pairs are sufficient for determining consensus number 1 objects. Also, such relationships can be utilized to implement these consensus number 1 objects with read/write registers for unbounded number of processes.

References

[1] Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., and Shavit, N. Atomic snapshots of shared memory. *Journal of the ACM (JACM) 40*, 4 (1993).

[2] Aspnes, J., Attiya, H., and Censor, K. Max registers, counters, and monotone circuits. In *Proceedings of the 28th ACM symposium on Principles of distributed computing (PODC)* (2009).

[3] Baudet, M., Danezis, G., and Sonnino, A. Fastpay: High-performance byzantine fault tolerant settlement. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies* (2020), pp. 163–177.

[4] Chang, J.-M., and Maxemchuk, N. F. Reliable broadcast protocols. *ACM Transactions on Computer Systems (TOCS) 2*, 3 (1984), 251–273.

[5] Ellen, F., Gelashvili, R., Shavit, N., and Zhu, L. A complexity-based hierarchy for multiprocessor synchronization. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing (PODC)* (2016).

[6] Guerraoui, R., Kuznetsov, P., Monti, M., Pavlovič, M., and Seredinschi, D.-A. The consensus number of a cryptocurrency. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (2019).

[7] Herlihy, M. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS) 13*, 1 (1991).

[8] Herlihy, M., Shavit, N., Luchangco, V., and Spear, M. *The art of multiprocessor programming*. Newnes, 2020.

[9] Herlihy, M. P., and Wing, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS) 12*, 3 (1990).

[10] Jayanti, P. On the robustness of herlihy's hierarchy. In *Proceedings of the twelfth annual ACM symposium on Principles of distributed computing* (1993).

[11] Khanchandani, P., and Wattenhofer, R. On the importance of synchronization primitives with low consensus numbers. In *Proceedings of the 19th International Conference on Distributed Computing and Networking (ICDCN)* (2018).

[12] Khanchandani, P., and Wattenhofer, R. Two elementary instructions make compare-and-swap. *Journal of Parallel and Distributed Computing (JPDC) 145* (2020).

[13] Nakamoto, S. Bitcoin: A Peer-to-Peer Electronic Cash System. 2008.

[14] Raynal, M. *Concurrent programming: algorithms, principles, and foundations*. Springer Science & Business Media, 2012.

[15] Ruppert, E. Consensus numbers of multi-objects. In *Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing* (1998).

[16] Ruppert, E. Consensus numbers of transactional objects. In *International Symposium on Distributed Computing* (1999), Springer.

[17] Ruppert, E. Determining consensus numbers. *SIAM Journal on Computing 30*, 4 (2000).

[18] Sliwinski, J., and Wattenhofer, R. Abc: Asynchronous blockchain without consensus. *arXiv preprint arXiv:1909.10926* (2019).