Does the operational model capture partition tolerance in distributed systems? (Extended Version)

Grégoire Bonin, Achour Mostefaoui, Matthieu Perrin

Does the operational model capture partition tolerance in distributed systems?

Grégoire Bonin, Achour Mostéfaoui, and Matthieu Perrin

LS2N, Université de Nantes
`first.last@univ-nantes.fr`

Abstract

In large scale distributed systems, replication is essential in order to provide availability and partition tolerance. Such systems are abstracted by the wait-free model, composed of asynchronous processes that communicate by sending and receiving messages, and in which any process may crash. Complexity in local memory has already been studied for several objects, including sets, databases and collaborative editors. However, the literature has focused on a subclass of algorithms, called the operational model, in which processes can only broadcast one message per update operation and the read operation incurs no communication.

This paper tackles the following question: are the operational model and the wait-free model equivalent from the complexity point of view?

We show that update consistency allows implementations in the wait-free model that require strictly less local memory than their counterparts in the operational model.

1   Introduction

In distributed systems, shared objects are used to simplify the implementation of distributed applications, and provide fault tolerance by replication. Problems arise with replication as consistency has to be maintained between the different replicas.

The most natural and intuitive abstraction for the user would be to view a distributed/replicated object as if it is a single physical object shared by all the processes. This means that all the operations on the object, possibly concurrent or interleaving, appear as if they have been executed atomically and sequentially. Such an abstraction has to respect a correctness condition called strong consistency. Unfortunately, the CAP Theorem [5] states that this property is unrealizable in most systems, as it is impossible to combine strong consistency, availability and partition tolerance in asynchronous systems. Eventual consistency was introduced to overcome this issue. It states that, after update operations stop taking place, the different replicas will eventual converge to an identical state.

In this context, Conflict-Free Replicated Data Types (CRDTs) [10] constitute a family of objects designed to achieve eventual consistency. Those are based on a theorem stating the equivalence between two kinds of objects: the Commutative Replicated Data Types (CmRDTs), in which all update operations commute, and Convergent Replicated Data Types (CvRDTs), whose states form a lattice. For example, the G-set (grow-only set) provides two different operations: an update operation that inserts an element and a query operation that reads if a specific element is in the set. On the CmRDT viewpoint, inserting x and inserting y commute. On the CvRDT viewpoint, the set inclusion is a lattice order on the states of the set. The operational model has been proposed to abstract the implementation of CRDTs. In the operational model, each replica maintains a local state on which the operations are done. An update operation is divided into two facets. First, the update operation is prepared locally by the replica where the update operation is issued and then a message is broadcast to inform all other replicas. Second, the local state of each replica is updated at reception of the update message. By the commutative effect all replicas converge to the same state when no update operation is in progress.

As only one message is broadcast per update operation, algorithms in the operational model are, by design, optimal in terms of the number of used messages. The amount of metadata that must be stored on each replica to ensure convergence is more problematic and has been widely studied for several objects including sets, counters and registers [4], data stores [2] and collaborative editors [1].

Despite the fact that algorithms from the operational model are naturally partition tolerant and minimize the number of messages needed in their implementation, the operational model imposes limitations on the form of its admissible algorithms. It is for example impossible to acknowledge or forward messages, to execute local steps without the reception of a message, or to propagate information during read operations. This prevents algorithms from using more advanced techniques like the message schemes used by checkpointing [8, 3]. Such algorithms are usually studied in the *wait-free asynchronous message-passing distributed model*, or simply the *wait-free model*, in which asynchronous processes communicate by sending and receiving messages. Any number of processes may crash, which also captures partition tolerance as it is impossible for a process to wait for an acknowledgement from any other process since all other processes may have crashed.

Problem statement The wait-free model is strictly more general than the operational model, as any algorithm from the operational model can be naturally transformed into an algorithm in the wait-free model, but the converse does not hold. In particular, this means that the complexity results proven in the operational model may not hold in the wait-free model.

Therefore arises the following question: are the wait-free model and the operational model equivalent in terms of complexity ?

Approach In this paper, we consider objects specified by a sequential speci-fication, that describes the behaviour of the object when processes access it sequentially, and a weak consistency criterion, namely update consistency [7]. Update consistency strengthens eventual consistency by stating that the conver-gence state must be obtainable in a sequentially consistent execution. In other words, it can be obtained by a sequential ordering of the update operations. On a computability viewpoint, it is possible to implement any object with this criteria in both computing models [7, **?**].

Contributions We prove that there exists an object whose implementation in the wait-free model requires strictly less local memory than its operational model counterpart.

In Section 2, we present the shared objects that we consider for our proofs, and more particularly a new object, the Countdown-append, as well as the two consistency criteria we use. In Section 3, we present the state of the art of the two models and define our formalism to study the properties of the implementations of the two models. In Section 4, we present the algorithm type and the complexity that we use to compare the different models. In Section 5, we prove the difference between the two models under update consistency. In Section **??**, we prove the equivalence of the two models under causal convergence.

2   Shared objects

In distributed systems, several kinds of shared objects have been proposed to provide the processes with higher-level abstractions. There are two main kinds of objects. On the one hand, one-shot objects like consensus and renaming are a generalization of functions in sequential systems, where each process proposes an input and decides an output. One-shot objects are specified by a binary relation that relates input vectors to the admitted output vectors. On the other hand, long-lived objects, such as registers and queues, are a generalization of data structures in sequential programming, aiming at storing and organizing data in memory. This paper only considers long-lived objects

Long-lived objects are defined by three components: a sequential specification that describes its expected behaviour when operations are accessed sequentially (e.g. a queue or a stack), a consistency criterion that describes how concurrency affects the object (e.g. linearizability or eventual consistency), and a progress condition that enforces termination guarantees. The only progress condition we address in this paper is wait-freedom: all operations invoked by non-faulty pro-cesses terminate.

2.1   Sequential specifications

In order to keep close enough to the CRDTs, we suppose that shared objects are accessed by two distinct kinds of operations: updates, from a set U, modify the local state of the object but do not return a value, and queries, from a set

Q, return a value in a set B that depends on the state of the object but do not modify it[1]. The set of all operations is $\mathsf{A} = U \cup (Q \times \mathsf{B})$. A *sequential history* is a (finite or infinite) sequence of symbols in A.

The *sequential specification* of an object is the set of all sequential histories admitted by the object. It describes the behaviour of the object when it is used by a single process.

As an example, we now define the *l*-countdown-append object, where $l \in \mathbb{N}$, that will be especially useful in Section 5. The *l*-countdown-append object exposes 4 update operations, a, b, c and d and one query operation, q. Figure 1 represents the behaviour of the object as an automaton. It is divided into two phases: during the first phase, the object counts the number of update operations, starting from l, down to 1, then ε (the empty word). In the second phase, the operation is concatenated at the end of the state. Finally, the query operation returns the local state of the objects each time it is executed.
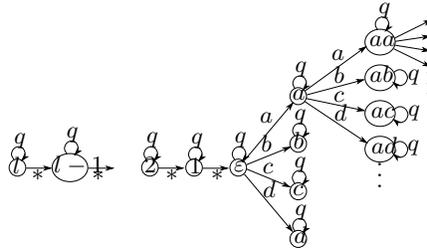

Fig. 1. Representation of a *l*-countdown-append object as an automaton.

2.2 Consistency criterion

A consistency criterion defines how concurrency affects the distributed behaviour of an object. Formally, it identifies which distributed histories are admissible for a given sequential specification.

A distributed history models a distributed executions of a program using a shared object. It is defined by a tuple $H = (A, E, L, \mapsto) \in \mathcal{H}$, where:

- A is the set of operations of the object (the same set as in sequential histories);
- E is a countable set of *events*;
- $L : E \mapsto A$ is a *labelling function*, that labels each event by an operation;
- $\mapsto \in E^2$ is the *process order*, a partial order such that $e \mapsto e'$ if e precedes e' on the same process.

---

[1] This hypothesis is only done for the sake of clarity and can be easily removed, like in [6].

A sequential history is a *linearization* of a distributed history H if it contains the same operations as H, and the order of appearance of the operations is compatible with the process order between the events labelled by the operations. We now define formally the consistency criterion used in this paper: update consistency.

Update consistency:

A history is update consistent [7] for an object O if, when all the processes stop executing update operations, they eventually converge towards a state resulting from a linearization of the update operations. Formally, a history H is update consistent if it is in one of the two following cases :

- The processes never stop updating, i.e. H contains an infinity of update operations.
- It is possible to omit a finite number of query operations such that resulting history has a linearization in the sequential specification of O.

3   Computing models

We now present the two computing models used in this paper: the wait-free model and the operational model, as well as the definition of an execution in the wait-free model.

3.1   Wait-free model

The *wait-free asynchronous message-passing system* model, or simply wait-free model, is composed of n processes called p_1, p_n. The set of all processes is denoted by P. The number n of participating processes is bounded, although it may not be known by each of them. Processes are asynchronous, in the sense that there is no bound on their relative speed. Moreover, processes can fail by crashing: a *faulty* process executes correctly until it *crashes*, at which point it stops operating. A process that does not crash during an execution is called *correct*.

Processes can communicate by sending and receiving messages. Communication channels are reliable, as all sent messages are eventually received by correct processes. However, channels are asynchronous, in the sense that there is no bound on the time it takes for one message to be delivered. We suppose that all sent messages can be uniquely identified.

We assume that processes have access to the *causal broadcast* abstraction that provides them with an operation **broadcast** (m) operation and a **receive** (m) event, where m is a message, respecting the following properties.

- Validity: If a process delivers a message m, then m was broadcast by some process.
- Uniformity: If a process delivers a message m, then all correct processes deliver m.

- Termination: If a correct process p_i attempts to broadcast m, then p_i terminates its broadcast invocation and eventually delivers m.
- Causal delivery: If a process delivers a message m and then broadcasts a message m', then all processes delivering m' have previously delivered m

Note that causal broadcast can be easily implemented in the wait-free model [9]. However, this implementation has a cost in local memory. We choose to include the primitive in the model to isolate the complexity needed to maintain consistency of the shared objects from the complexity needed to ensure complexity, and therefore reducing the noise of the complexity results we obtain in the next sections.

3.2   Operational model

The operational model allows algorithms of form of Algorithm 1. Each process maintains a local state. Query operations return a value that is locally computed based on the local state. Update operations are separated into a *prepare* function and an *effect* function. The prepare function computes locally a piece of information m based on the update function and the local state. Then, function $effect(m)$ is applied asynchronously on all processes.

```
1  object A
2  │    Payload
3  │    └   description of the local state
4  │    Query q
5  │    └   return δ_q(Payload)
6  │    Update u
7  │    │    prepare
8  │    │    └   m is an information to be passed
9  │    │    effect(m)
10 │    │    └   Payload ← τ_u(m, Payload)
```

Algorithm 1: operational model algorithmic form

There is a canonical injection that maps any algorithm A in the operational model into an algorithm $op2wf(A)$ in the wait-free model, as illustrated by Algorithm 2. The local states of $op2wf(A)$ are the same as the payload of A and the queries remain unchanged. In the update operations, the result of the *prepare* function is transmitted to the *effect* function as a single message forwarded on the network. Because of this, the operational model can be viewed as a restriction of the wait-free model, where additional constraints on the use of the messages have been added.

By a slight abuse of language, from now on, we will use the term "algorithm" (without precision), to refer to algorithms in the wait-free model. Further notions

defined on algorithms are extended to algorithms from the operational model thanks to the canonical injection.

```
1  object A ∈ OM(T)
2  │   local_A ← ζ_0;
3  │   operation α is
4  │   │   if α ∈ U_T then
5  │   │   └   Send(M(α));
6  │   │   Return δ_α(local_A);
7  │   when a message M(α) is received from
8  │   └   local_A ← τ_α(local_A);
```

Algorithm 2: Wait-free model translation of the operational model algorithmic form

4   Complexity

We consider deterministic algorithms. This allows us to define a state by an execution or a history. In order to compare the local complexity of algorithms in the different models, we define the *H-complexity* that allows us to compare the efficiency of two algorithms when executing the same history. As the algorithms are deterministic, we can compare equivalent state in the two algorithms (if the states are defined by the same sub-history, then they are equivalent).

Definition 1 (*H*-complexity). *Let H be a history that contains a finite number of updates, and let Λ be an algorithm. Let S be the set of all local states reachable by any process executing Λ during an execution that can be abstracted by H.*

We define the H-complexity of Λ as follows:

- *if $S = \emptyset$ (i.e. if H is not admitted by Λ), the H-complexity is 0;*
- *if S is infinite (i.e. if S contains states of unbounded size), the H-complexity is ∞;*
- *otherwise, the H-complexity is the maximal size of a state in S.*

5   Update consistent countdown-append object

In this section, we prove that the two models are not equivalent under update consistency. To do so, we study implementations of the *l*-Countdown-append object in both models. More precisely, we compare the number of bits that can

be needed to encode the local state of some process after it executes l update operations.

For all words $v = u_1...u_l \in \{U^l\}$ consisting of l update operations of the l-countdown-append object, we denote by $H_v = (U, \{1, ..., l\}, i \to u_i, < |\{1, ..., l\})$ the history in which one process performs all updates of v in their order of appearance.

We prove the following results. On the one hand, the H_v-complexity of any algorithm in the operational model is at least $\frac{l}{2} - 1$ bits, for some v. On the other hand, there exists an algorithm in the wait free model with an H_v-complexity in $\mathcal{O}(n \log(nl))$ bits for all v.

Intuitively, the reason for this difference comes from the fact that the information returned by the query operation changes between the countdown phase and the append phase of the object. Because of message reordering, it happens that the operations that form the return value in the second phase are already known (and misinterpreted) by some processes during the first phase. In the wait-free model, this information can be retrieved from other processes afterwards. This is not possible in the operational model, so processes have to store it locally.

5.1   Lower bound in the operational model

We now prove that any algorithm in the operational model has a H_v-complexity of at least $\frac{l}{2} - 1$ bits for some v. Our proof follows the scheme introduced in [4]: we build a family of executions such that, at some point in the execution, process p_i performing the operations of v is unable to distinguish between all these executions and an execution modeled by H_v. Then, in a later stage of the execution, p_i must be able to distinguish between enough of them in order to keep convergence possible.
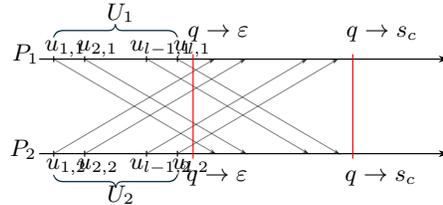
Fig. 2. Typical execution in the proof of Theorem 1

Theorem 1. *For any deterministic algorithm Λ that implements an update consistent l-countdown-append object in the operational model, there exists v such that the H_v-complexity of Λ is at least $\frac{l}{2} - 1$ bits.*

Proof. Let Λ be an algorithm in the operational model implementing an update consistent l-countdown-append object. For each pair of words of update

operations (v_1, v_2), where $v_1 \in \{a, b\}^l$ and $v_2 \in \{c, d\}^l$, we define the execution $X_{(v_1, v_2)}$, illustrated on Figure 2, as follows. Only two processes p_1 and p_2 take steps in $X_{(v_1, v_2)}$. All other processes crash before the beginning of the execution. Initially, process p_1 (resp. p_2) executes sequentially, in order, the operations forming v_1 (resp. v_2). In accordance to the operational model, they broadcast a single message during each operation. In a later stage, they both receive the others' messages, respecting the FIFO ordering. Finally, both processes perform a query operation. We denote by $\mathcal{X} = \{X_{(v_1, v_2)} | v_1 \in \{a, b\}^l \wedge v_2 \in \{c, d\}^l\}$ the set of all $X_{(v_1, v_2)}$ executions.

Let us first remark that update consistency imposes that both query operations returns the same value v_c, that is a suffix of size l, of an interleaving of v_1 and v_2. Let $f(v_1, v_2)$ be the number of c and d operations in v_c. Note that f is well defined because Λ is deterministic.

We now distinguish the executions depending on which process has a majority of operations in the convergence state. We define $\mathcal{X}_1 = \{X_{(v_1, v_2)} \in \mathcal{X} : f(v_1, v_2) \geq \frac{l}{2}\}$ and $\mathcal{X}_2 = \mathcal{X} \setminus \mathcal{X}_1$. As $\mathcal{X}_1$ and $\mathcal{X}_2$ form a partition of $\mathcal{X}$ which has a size 2^{2l}, we have $|\mathcal{X}_1| \geq 2^{2l-1}$ or $|\mathcal{X}_2| \geq 2^{2l-1}$. Without loss of generality, we suppose that $|\mathcal{X}_1| \geq 2^{2l-1}$.

We now partition $\mathcal{X}_1$ based on the value of v_1. For each word $v_1 \in \{a, b\}^l$, let $\mathcal{X}_1(v_1) = \{X_{(v, v_2)} \in \mathcal{X}_1 : v = v_1\}$. There exists a word v_1 such that $|\mathcal{X}_1(v_1)| \geq \frac{|\mathcal{X}_1|}{|\{a,b\}^l|} = \frac{2^{2l-1}}{2^l} = 2^{l-1}$. Let us fix such a v_1.

Let v_2 and v_2' such that $X_{(v_1, v_2)}$ and $X_{(v_1, v_2')}$ belong to $\mathcal{X}_1(v_1)$. By definition of f, if $X_{(v_1, v_2)}$ and $X_{(v_1, v_2')}$ converge to the same state, then v_2 and v_2' differ at most by their $l - f(v_1, v_2) \leq \frac{l}{2}$ first operations. Consequently, there are at least $\frac{2^{l-1}}{2^{\frac{l}{2}}} = 2^{\frac{l}{2}-1}$ different values for v_2 for which $X_{(v_1, v_2)}$ lead to different convergence states. Let $\mathcal{X}'$ be a subset of $\mathcal{X}_1(v_1)$ of size $2^{\frac{l}{2}-1}$, in which all convergence states are different.

In the operational model, the local state of process p_2 at the end of the execution only depends on its local state after executing its own l update operations, and the messages received from p_1 afterwards. In all the executions of $\mathcal{X}'$, the messages received by p_2 are the same in all executions because v_1 is fixed. Moreover, the local state of p_2 at the end of all executions is different. This means that the local state of p_2 after doing its updates is also different in all executions. Consequently, there is a word v_2 such that, after executing all update operations in v_2 (execution X), the local state of p_2 requires at least $\frac{l}{2} - 1$ bits.

Finally, let us consider the execution X' in which only p_2 takes steps, executing a the sequence of update operations of v_2. Just after executing its updates, p_2 cannot distinguish between executions X and X', so its local state in X' also requires $\frac{l}{2} - 1$ bits. Moreover, X' is modeled by H_{v_2}. Therefore, the H_{v_2}-complexity of Λ is at least $\frac{l}{2} - 1$ bits.

5.2   Upper bound in the wait-free model

We now prove there is an algorithm that implements an update consistent l-Countdown-append in the wait-free model with a lower H_v-complexity, for any v. Our proof is based on Algorithm 3, based on the algorithm UQ_0 from [6].

Each process p_i maintains four variables. Variables $\mathsf{countdown}_i$ and append_i represent the current local state at p_i. If $\mathsf{countdown}_i > 0$, the l-countdown-append object is in the countdown phase. Otherwise it is in the append phase and its value is append_i. Variable clock_i is the equivalent of a version vector, such that $\mathsf{clock}_i(j)$ that represents the number of operations done by p_j that are taken into account into the current state of p_i. As p_i does not know the number of participants, it is encoded as an associative array, rather than a vector. Finally, variable leader_i is the identifier of a process such that, $\mathsf{clock}_i < \mathsf{clock}_{\mathsf{leader}_i}$ or p_i and p_{leader_i} are in the same local state.

When a process invokes the query operation q, it computes locally the state of the object based on $\mathsf{countdown}_i$ and append_i.

When process p_i invokes an update operation a, b, c or d, it increments its local clock $\mathsf{clock}_i[i]$ and broadcasts a message $\mathtt{mUpdate}$ (Line 9). At reception of such a message, p_i executes the operation (decrements $\mathsf{countdown}_i$ if the countdown is not finished, or append the operation to append_i), and answers with a $\mathtt{mUpdate}$ message containing its version of the state and its current vector clock.

When receiving a correction message, the process checks if the message received is more recent according to the vector clock, and if that is the case, it replaces its own data with the received one.

In order to keep the paper as readable as possible, we only give a sketch of the correctness proof here. A more formal proof can be found in [6].

Lemma 1 (Update consistency). *All distributed histories admitted by Algorithm 3 are update consistent for the l-countdown-append object.*

Proof. Let H a history admitted by Algorithm 3. If H contains a finite number of queries or an infinite number of updates, it is update consistent by definition. Otherwise, at least the process performing updates is correct. Let us suppose that each process p_i performs a finite number m_i of updates. Let us pose $m = \sum_{i=1}^{n} m_i$. After some finite time t_1, all correct processes have received all messages $\mathtt{mUpdate}$ sent during the execution, and after some finite time t_2, all correct processes have received all messages $\mathtt{mCorrect}$ sent during the execution.

If $m \leq l$, after t_1, all read operations return $l - m$ (or ε when $l = m$), which is correct for update consistency. If $m > l$, all correct processes broadcast $m - l$ messages $\mathtt{mCorrect}$. Let us consider the process p_j that broadcast $m-l$ messages $\mathtt{mCorrect}$, with the smallest identifier (it exists because at least one process is correct), and let $mC = \mathtt{mCorrect}(cl_j, j, a_j)$ be its last message. Because of the causal broadcast, no process can receive a message $\mathtt{mUpdate}$ after mC. Moreover, the clock cl_j is maximal and j is minimal, so by the condition of lines 18-19, all processes will adopt state a_j at reception of mC and keep it afterwards. Finally, a_j is corresponds to a state accessible by a linearization of the updates in H: the clock mechanism of lines 11 and 18-19 ensures that each update is applied

```
 1  var clock_i ∈ Array(ℕ, ℕ) ← [i ↦ 0];
 2  var leader_i ∈ ℕ ← i;
 3  var countdown_i ∈ {0, ..., l} ← l;
 4  var append_i ∈ U* ← ε;
 5  operation q()
 6  │    if countdown_i = 0 then  return append_i;
 7  │    else  return countdown_i;

 8  operation u() //  u ∈ U
 9  │    broadcast mUpdate (clock_i[i] + 1, i, u);

10  receive mUpdate (t_j ∈ ℕ, j ∈ ℕ, u ∈ U)
11  │    if clock_i[j] < t_j then
12  │    │    clock_i[j] ← t_j; leader_i ← i;
13  │    │    if countdown_i = 0 then
14  │    │    │    append_i ← append_i · u;
15  │    │    │    broadcast mCorrect (clock_i, i, append_i);
16  │    │    else  countdown_i ← countdown_i − 1;

17  receive mCorrect (cl_j ∈ Array(ℕ, ℕ), j ∈ ℕ, a_j ∈ U*)
18  │    if (∀k, clock_i[k] ≤ cl_j[k])∧
19  │           (j ≤ leader_i ∨ ∃k, clock_i[k] < cl_j[k]) then
20  │    │    append_i ← a_j; clock_i ← cl_j; leader_i ← j;
```

Algorithm 3: The countdown-append object in the wait-free model

exactly once, and causal delivery ensures that the order between the updates is respected.

Lemma 2 (Wait-freedom). *Algorithm 3 is wait-free.*

Proof. This comes from the fact that it contains no loop.

Lemma 3 (Complexity). *For all $l \in \mathbb{N}$ and $v \in U^l$, Algorithm 3 has an H_v-complexity of $\mathcal{O}(\log(nl))$ bits.*

Proof. Let $l \in \mathbb{N}$ and $v \in U^l$. In any execution abstracted by H_v, there is a process p_i that performs all l update operations, and sends all messages mUpdate. At reception of any of these messages by any process p_j, the condition of Line 13 is false, so no message mUpdate is ever sent.

For all processes p_j and $p_k \neq p_i$, $\mathsf{clock}_j[k] = 0$ and $\mathsf{clock}_j[i] \leq l$. If we encode clock_i by a set of pairs $\langle j, \mathsf{clock}_i[j] \rangle$, the encoding takes less than $\log(n) + \log(l) = \log(nl)$ bits; The process identifier leader_i can be encoded in $\log(n)$ bits; $\mathsf{countdown}_i$ can take at most l different values, so it can be encoded in $\log(l)$ bits; finally $\mathsf{append}_i = \varepsilon$ is a constant value, so it has an encoding of constant size c.

Theorem 2. *There exists an algorithm Λ implementing an update consistent l-countdown-append object in the wait-free model such that, for all $v \in U^l$, Λ has an H_v-complexity of $\mathcal{O}(\log(nl))$ bits.*

Proof. By Lemmas 1, 2 and 3, Algorithm 3 is an example.

We can finally conclude on the non-equivalence between the two computing model in the implementation of update consistency.

Corollary 1. *There exists an object O and an algorithm Λ_{wf} implementing an update consistent O in the wait-free model, such that, for any algorithm Λ_{om} implementing an update consistent O object in the operational model, there is a history H such that Λ_{wf} has a strictly lower H-complexity than Λ_{om}.*

Proof. By Lemma 3, there exists a constant $x > 1$ such that, for any v, the H_v complexity of Algorithm 3 is strictly lower than $x \log(nl)$ bits. Let $l > 2^{-\frac{1+x \log n}{x}}$. We have $x \log(nl) < \frac{l}{2} - 1$.

Let O be the l-countdown-append object, and let Λ_{om} implementing an update consistent O in the operational model. By Theorem 1, there exists v such that the H_v-complexity of Λ_{om} is at least $\frac{l}{2} - 1$ bits. Therefore, Algorithm 3 has a strictly lower H_v-complexity than Λ_{om}.

6   Conclusion

In this paper we answered the following question: are the wait-free model and the operational model equivalent in terms of local complexity? We proved that the response to this question is no in the case of update consistency: we proved that there exists an object that has a different complexity in the two models: the l-countdown-append object. In the wait-free model, there is an algorithm for which the complexity required to encode a special state of the object is upper bounded by $\mathcal{O}(\log(nl))$ bits, whereas in the operational model, any algorithm requires at least $\frac{l}{2} - 1$ bits to encode the same state. This means that the operational model does not allow the optimal implementation for update consistency.

The result proposed in this papers shows that the question of whether the operational model is well suited to represent partition tolerance is not simple, especially in the context of determining the complexity in local memory required to implement shared objects. An interesting open question is whether the lower bounds proved for several objects in the operational model can be extended to the wait-free model.

References

1. Attiya, H., Burckhardt, S., Gotsman, A., Morrison, A., Yang, H., Zawirski, M.: Specification and complexity of collaborative text editing. In: Symposium on Principles of Distributed Computing. pp. 259–268. ACM (2016)
2. Attiya, H., Ellen, F., Morrison, A.: Limitations of highly-available eventually-consistent data stores. IEEE Trans. Parallel Distrib. Syst. **28**(1), 141–155 (2017)
3. Baldoni, R., Brzezinski, J., Hélary, J.M., Mostefaoui, A., Raynal, M.: Characterization of consistent global checkpoints in large-scale distributed systems. In: Workshop on Future Trends of Dist. Computing Systems. pp. 314–323. IEEE (1995)

4. Burckhardt, S., Gotsman, A., Yang, H., Zawirski, M.: Replicated data types: specification, verification, optimality. In: ACM Sigplan Notices. vol. 49, pp. 271–284. ACM (2014)
5. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. Acm Sigact News (2002)
6. Perrin, M.: Distributed Systems: Concurrency and Consistency. Elsevier (2017)
7. Perrin, M., Mostefaoui, A., Jard, C.: Update consistency for wait-free concurrent objects. In: International Parallel and Distributed Processing Symposium. pp. 219–228. IEEE (2015)
8. Randell, B., Lee, P., Treleaven, P.C.: Reliability issues in computing system design. ACM Computing Surveys (CSUR) **10**(2), 123–165 (1978)
9. Raynal, M., Schiper, A., Toueg, S.: The causal ordering abstraction and a simple way to implement it. Information processing letters **39**(6), 343–350 (1991)
10. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In: Symposium on Self-Stabilizing Systems. pp. 386–400. Springer (2011)