Complexity of Update Consistent Universal Constructions

Grégoire Bonin, Achour Mostefaoui, Matthieu Perrin, Olivier Ruas

Complexity of Update Consistent Universal Constructions

Grégoire Bonin[†] Achour Mostéfaoui[†] Matthieu Perrin[†] Olivier Ruas[‡]
[†] LS2N, Université de Nantes, 44322 Nantes Cedex, France
[‡] Peking University, Beijing, P.R. China

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. The CAP theorem states that strong consistency is unachievable in the wait-free model. Weaker consistency criteria, such as eventual consistency, update consistency and causal convergence, have been identified as potential substitutes to strong consistency for the management of replicated objects. 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 eventual consistency allows implementations in the wait-free model that require strictly less local memory than their counterparts in the operational model. On the other side, we propose, for the wait-free model, a universal construction that provides a garbage collection mechanism for old messages that never violates consistency even when there is no bound on the relative transmission delays.

Keywords: Consistency Criteria, Eventual Consistency, Replicated Object, Sequential Consistency, Space Complexity, Universal Construction, Update Consistency.

1 Introduction

Reliability of large scale systems is a big challenge when building massive distributed applications over the Internet. At this scale, data replication is essential to ensure availability and fault-tolerance. In a perfect world, distributed objects should behave as if there is a unique physical shared object that evolves following the atomic operations issued by the participants[1]. This means that all the operations on the object, possibly concurrent or interleaving, appear as if they have been executed atomically and sequentially. This is the aim of strong consistency criteria such as linearizability and sequential consistency. These criteria serialize all the operations so that they look as if they happened sequentially, but they are costly (when not impossible) to implement in message-passing systems. If one considers a distributed implementation of a shared register, the

[1]We use indifferently participant or process to designate the computing entities that invoke the operations of the distributed object.

worst-case response time must be proportional to the uncertainty on the latency of the network either for the reads or for the writes to be sequentially consistent [1] and for all the operations for linearizability [2]. This generalizes to many objects [2]. Moreover, the availability of the shared object cannot be ensured in asynchronous systems where more than a minority of the processes of a system may crash [3] (which we call the *wait-free model*), or when partitions may occur, isolating some processes from the others. In large modern distributed systems such as Amazon's cloud, partitions do occur between data centers, as well as inside data centers [4]. Moreover, it is economically unacceptable to sacrifice availability. The only solution is then to provide weaker consistency criteria. Several weak consistency criteria have been considered for modeling shared memory such as PRAM [1] or causality [5]. They expect the local histories observed by each process to be plausible, regardless of the other processes. However, these criteria do not impose that the data eventually converges to a consistent state. Eventual consistency [4] is another weak consistency criterion which was introduced to overcome this issue. It states that, after update operations stop taking place, the different replicas will eventually converge to an identical state. The relevance of eventual consistency has been illustrated many times. It is used in practice in many large scale applications such as Amazon's Dynamo highly available key-value store [6]. It has been widely studied and many algorithms have been proposed to implement eventually consistent shared object.

In this context, Conflict-Free Replicated Data Types (CRDTs) [7] 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 belongs to the set. On the CmRDT point of view, inserting x and inserting y commute. On the CvRDT point of view, the set inclusion is a lattice order on the states of the set. Unfortunately, many useful objects are not CRDTs.

This paper focuses on universal constructions of eventually consistent shared objects. The behavior of an object (counter, stack, ...) is described by a sequential specification. A universal construction is an algorithm that is parametrized by the sequential specification of a data type and automatically transforms it into a concurrent data structure respecting a given consistency criterion. Unfortunately, eventual consistency requires the convergence towards a *common state* without specifying which states are legal. This means that an algorithm that always leaves the shared object in its initial state would, indeed, be a (useless) eventually consistent universal construction. The limitations of eventual consistency led to the study of stronger criteria such as update consistency [8]. Update consistency strengthens eventual consistency by stating that the convergence state must be obtainable in a sequentially consistent execution. In other words, it can be obtained by a sequential ordering of the update operations.

Problem statement This paper explores the following issue: what is the complexity of update consistent universal constructions in partition-prone systems? Two metrics are relevant: the number of messages that are sent for each operation, and the size of the metadata stored at each replica.

The *operational model* has been proposed to abstract the implementation of CRDTs. In the op-

erational model, each replica maintains a local state on which the operations are done. An update operation is divided into two parts. First, the update operation is prepared locally by the replica where the update operation is issued and a message is broadcast to inform all the other replicas. Second, the local state of each replica is updated upon the reception of the update message. Thanks to commutativity, all replicas converge to the same state when no update operation is in progress. As only n messages are sent by each update operation, the operational model naturally leads to algorithms that are optimal in the number of sent messages.

On a computability point of view, update consistent universal constructions have already been proposed in the operational model [8, 9]. The key idea of these algorithms is to order the operations *a priori* using a timestamp given during the preparation phase, and to keep a log of all operations ordered according to these timestamps in the update phase. When two processes have received the same set of messages, they agree on their respective log, so they converge to the same state as well. The main problem of this approach is the size of the log, that keeps increasing as new operations are invoked. One can advocate that even though the system is asynchronous, it is very unlikely that the transfer delay of a message exceeds for example one day and consequently, all "very old" states can be garbaged as no old message can force to re-execute newer operations. This may work but is not safe, as if, for some reason, a message transmission exceeds this maximal assumed delay, convergence is no more guaranteed. Hence the following questions: is it possible to safely prune the log from very old operations, and, if yes, at which cost?

Contributions of the paper This paper has two main contributions.

- This paper first proves that the answer is "no" in the operational model, but "yes" in the wait-free model. For that we introduce an object called l-countdown-append, where l is an integer parameter. We prove that $\mathcal{O}(l)$ bits are necessary in the operational model to implement an update consistent l-countdown-append object. On the other side, we give a logarithmic space complexity algorithm in the wait-free model. This contribution has deep theoretical implications, as it proves that the two models are not equivalent regarding complexity, which questions the relevance of already known complexity results regarding CRDT algorithms.
- Unfortunately, the logarithmic algorithm presented in the first contribution for the wait-free model requires $O(n^2)$ messages per operation. We propose a practical trade-off, a universal construction called $UC[k]$, that only requires n messages per operation when there is a bound k and provides a garbage collection mechanism for old messages that is safe even when the bound is violated. The parameter k reports on the relationship between the relative transmission delay of messages and the number of issued update operations during the transmission of one message.

Organization of the paper The remainder of this paper is organized as follows. In Section 2, we define more formally the concepts and objects considered in this paper. Section 3 presents more precisely the wait-free and operational computing models. Section 4 recalls the universal construction UC_∞ in the operational model, that was introduced in [8]. Section 5 proves the lower bound in the operational model and gives an upper bound in the wait-free model, proving that

the two models are not equivalent when we consider complexity. Section 6 presents the practical universal construction $UC[k]$. Finally, Section 7 concludes the paper.

2 Update consistent universal constructions

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, to concurrent systems, of the concept of function 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.

A long-lived object is defined by three components: a sequential specification that describes its expected behavior when accessed sequentially (queue, stack, ...), a consistency criterion that describes how concurrency affects the object (linearizability, eventual consistency, ...), and a progress condition that enforces liveness guarantees. The only progress condition we address in this paper is wait-freedom: all operations invoked by non-faulty processes, terminate regardless of the failure pattern and the message schedule.

2.1 Sequential specifications

A long-lived object exposes operations that can be invoked by processes and may return a value. The *sequential specification* of an object is the set of all *sequential histories* admitted by the object, i.e. the finite or infinite sequences of operation invocations and responses that can be produced when all the operations on the object are issued by a unique sequential process.

A universal construction is an algorithm that exposes a single operation, `apply`, and that can simulate the operations of any object whose sequential specification is described by an *abstract data type* as an automaton [9]. An abstract data type is a tuple $(\mathcal{O}, \mathcal{R}, \mathcal{S}, s_0, u, q)$ where:

- $\mathcal{O}$ is a countable set containing all the operations on the object, and that can be passed as argument to the operation `apply`. If an operation has parameters, each combination of different parameters is abstracted as a different symbol in $\mathcal{O}$.
- $\mathcal{R}$ is a countable set containing all the values that can be returned by operations on the object, as well as by the operation `apply`. $\mathcal{R}$ may possibly include dummy values for operations that do not have a return value.
- $\mathcal{S}$ is a countable set of *states* and $s_0 \in \mathcal{S}$ is the *initial state*;
- $u : \mathcal{S} \times \mathcal{O} \to \mathcal{S}$, called the *update* function, encodes the side effects of an operation on the current state of the object. An *update operation* is an operation o for which the update function is not the identity, i.e. for some state s, $u(s, o) \neq s$.

- $q : \mathcal{S} \times \mathcal{O} \to \mathcal{R}$ is the *query* function, that encodes the value returned by an operation, depending on the state on which it is invoked. A *query operation* is an operation whose return value depends on the state on which it is performed, i.e. there exist two states $s \neq s'$ such that $q(s, o) \neq q(s', o)$.

Remark that, in the most general case, an operation may be both an update and a query operation. For example, the *pop* operation on a stack removes the last element added to the stack (its update part), and also returns it (its query part).

2.2 Consistency

A consistency criterion defines how concurrency affects the distributed behavior of an object. Formally, it identifies which distributed histories are admissible for a given sequential specification. A distributed history models a distributed execution of a program accessing a shared object. It is composed of a (finite or infinite) set of events labelled by the operations of the object (or the operations passed as arguments to `apply` in the case of a universal construction) and their return values. This set is ordered by the process order, a partial order such that $e \mapsto e'$ if e and e' have been executed by the same process in that order.

A sequential history is a *linearization* of a distributed history H if it contains the same operations and returned values as H, and the order of appearance of the operations, in the sequential history, does not contradict the process order defined above. We now define formally the consistency criterion used in this paper: update consistency.

Update consistency A history is update consistent [8] for an object O if, when all the processes stop executing update operations, they eventually converge towards a state resulting from a linearization of all issued update operations. Formally, a history H is update consistent if it falls under one of the following two cases.

- The processes never stop updating, i.e. H contains an infinity of update operations.
- It is possible to change the returned value of a finite number of query operations, so that the 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.

3.1 Wait-free model

The *wait-free asynchronous message-passing system* model, or simply *wait-free model*, is composed of n processes called $p_1, \ldots p_n$. The number n of participating processes is finite, although it may not

be known to the processes. 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*, and then 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.

Remark that the wait-free model also captures partition tolerance because a process cannot wait for an acknowledgment from any other process, since they may all have crashed.

We assume that processes have access to a *causal broadcast* abstraction that provides them with a **broadcast** m operation and a "**when a message** m **is received from** p_j" event, where m is a message and p_j is a process, respecting the following properties.

Validity. If a process receives a message m from p_j, then m was broadcast by p_j.

Uniformity. If a process receives a message m, then all correct processes receive m.

Termination. If a correct process p_i attempts to broadcast m, then p_i terminates its broadcast invocation and eventually receives m.

Causal delivery. If a process receives a message m and then broadcasts a message m', then all processes receiving m' have previously received m.

Note that causal broadcast can be implemented in the wait-free model [10] without additional computing power. 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 causality, and therefore reducing the noise on the complexity results we obtain in the next sections.

3.2 Operational model

The *operational model* is composed of n replicas $r_1, \ldots, r_n$, where $n \in \mathbb{N}$ may not be known to each of them.

All replicas execute the same algorithm following a very specific format, which defines the model, presented thereafter.

Each replica maintains a local state, called its payload, and can interact with the system by invoking update or query operations. A query operation o returns a value $\texttt{query}_o(\mathsf{payload})$ that is locally computed based on the local state of the replica. An update operation o is separated into a $\texttt{prepare}_o$ function and an $\texttt{effect}_o$ function. The $\texttt{prepare}_o$ function computes locally a piece of information data based on the update function and the local state. Then, $\texttt{effect}_o(\mathsf{data}, \mathsf{payload})$ is applied asynchronously on the local state of all replicas. It is required that all $\texttt{effect}_o$ functions commute, so that all replicas eventually converge to a common state, ensuring eventual consistency.

Algorithm 1: Wait-free model translation of an operational model algorithm

```
1 operation apply(o ∈ 𝒪) ∈ ℛ
2     if o is an update then
3         broadcast mUpdate(o, prepare_o(payload_i));
4     return query_o(payload);
5 when a message mUpdate(o_j, x_j) is received from p_j
6     payload_i ← effect_{o_j}(x_j, payload_i);
```

Algorithms in the operational model can be seen as a special case of algorithms in the wait-free model. More precisely, Algorithm 1 is a canonical injection that maps any algorithm in the operational model into an algorithm in the wait-free model. Each replica from the operational model is embedded on a process in the wait-free model, that maintains the payload as its local state. When a process p_i performs an operation o of the operational model, it invokes the operation $\texttt{apply}(o)$ of Algorithm 1. Process p_i evaluates queries operations using the same algorithm $\texttt{query}_o$, applied on its local state (Line 4). If o is an update operations, the result of the $\texttt{prepare}_o$ function is transmitted to the $\texttt{effect}_o$ function as a single message broadcast on the network (Line 3).

Thanks to this transformation, 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: a message can only be broadcast on the network when an update operation is issued. 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, and we will say that an algorithm in the wait-free model belongs to the operational model if it is an image of an algorithm in the operational model by the canonical injection, i.e. if it respects the constraints on message broadcasts. Further notions defined on algorithms are extended to algorithms from the operational model thanks to the canonical injection.

4 A universal construction in the operational model

We now recall the update consistent universal construction in the operational model, that was introduced in [8]. The code of Algorithm 2 is given for process p_i.

The principle is to build a total order on the updates on which all the participants agree *a priori*, and then to rewrite the history *a posteriori* so that every replica of the object eventually reaches the state corresponding to the common sequential history. In Algorithm 2, this order is built from a Lamport's clock [11] that contains the happened-before precedence relation. In order to have a total order, the events are timestamped with a pair composed of the logical time and the identifier of the process that produced it.

Each process p_i manages its view time_i of the logical clock and a list $\mathsf{history}_i$ of all timestamped update events process p_i is aware of. The list $\mathsf{history}_i$ contains triplets (t, j, o) where o is an update operation and (t, j) the associated timestamp. This list is sorted according to the timestamps of the updates: $(t, j) < (t', j')$ if ($t < t'$) or ($t = t'$ and $j < j'$).

Algorithm 2: Universal construction $UC_\infty(\mathcal{O}, \mathcal{R}, \mathcal{S}, s_0, u, q)$: code for p_i

```
1 operation apply(o ∈ O) ∈ R
2     var s ∈ S ← s_0;
3     if o is a query then
4         for (t, j, o') ∈ history_i sorted according to (t, j) do s ← u(s, o') ;
5     if o is an update then time_i ← time_i + 1; broadcast mUpdate (time_i, o) ;
6     return q(s, o);
7 when a message mUpdate (t_j ∈ N, o_j ∈ O) is received from p_j
8     time_i ← max(time_i, t_j) ;
9     history_i ← history_i ∪ {(t_j, j, o_j)};
```

When an update is issued locally, process p_i timestamps it and informs all the other processes by reliably broadcasting a message to all other processes (including itself), on Line 5. Hence, all processes will eventually be aware of all updates. When a $\texttt{mUpdate}(t_j, o_j)$ message is received, p_i updates its clock and inserts the event into the list $\mathsf{history}_i$ (Line 7). When a query is issued, p_i replays locally the whole list of update events it is aware of starting from the initial state then it executes the query on the state it obtains (Line 4).

In an execution that contains only a finite number of update operations, then eventually all processes will be aware of the same set of updates and sort them in the same order to evaluate their query operations. This implies update consistency. Whenever an operation is issued, its is completed without waiting for any other process. This corresponds to wait-free executions in shared memory distributed systems and implies fault-tolerance.

5 Comparison of the computing models

Algorithms from the operational model are naturally partition tolerant and, as only one message is broadcast per update operation, they are by design optimal in terms of the number of sent messages. However, 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 message patterns used by checkpointing protocols [12, 13].

Following our quest to an efficient universal construction, we study the impact that these limitations have on the memory complexity of concurrent algorithms. Indeed, the amount of metadata that must be stored on each replica to ensure convergence in the operational model is problematic and has been widely studied for several objects including sets, counters and registers [14], data stores [15] and collaborative editors [16].

In this section, we prove that the operational model is not equivalent to the wait-free model in terms of memory complexity. More precisely, we exhibit a family of objects, called *l-countdown-append* objects, and we prove that any update consistent algorithm implementing them in the

operational model requires at least $O(l)$ bits of metadata in some specific executions. On the other hand, only a logarithmic number of bits are required in the wait-free model.

This result has an impact on the complexity of universal constructions. The linear lower bound proven for the l-countdown-append object in the operational model is similar to the upper bound given on universal constructions by Algorithm 2. In particular, it suggests that it is impossible to safely prune the log of from very old operations, while remaining in the operational model. This is nevertheless possible in the wait-free model, at the cost of an increased number of messages.

On a theoretical point of view, this result also questions the generality of the complexity studies of various CRDTs mentioned above, since these studies only take into account algorithms in the operational model.

5.1 Complexity of update consistent l-countdown-append objects

Our proof of non-equivalence compares the number of bits necessary to encode the local states of processes when executing a specific pattern of operations on a specific family of objects. We first need to precisely specify our notion of complexity and define *l-countdown-append* objects.

Complexity of deterministic algorithms Executions of different algorithms in the wait-free model can be very different, so we can only compare algorithms based on distributed histories, that form the basis of their specification. We define the *H-complexity* as the maximal size of a local state reachable during an execution abstracted by the history H.

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

We define the H-complexity of A as follows:

- if $S = \emptyset$ (i.e. if H is not admitted by A), 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.

Countdown-append object The l-countdown-append object, where $l \in \mathbb{N}$, exposes 4 update operations, a, b, c and d, and one query operation q. Figure 1 represents the behavior 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 operations are concatenated at the end of the state. Finally, the query operation returns the local state of the objects each time it is executed.

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 the distributed history in which one process performs all updates of v in their order of appearance in v. In the next sections, we study the H_v-complexity of algorithms in the operational model and in the wait-free model.

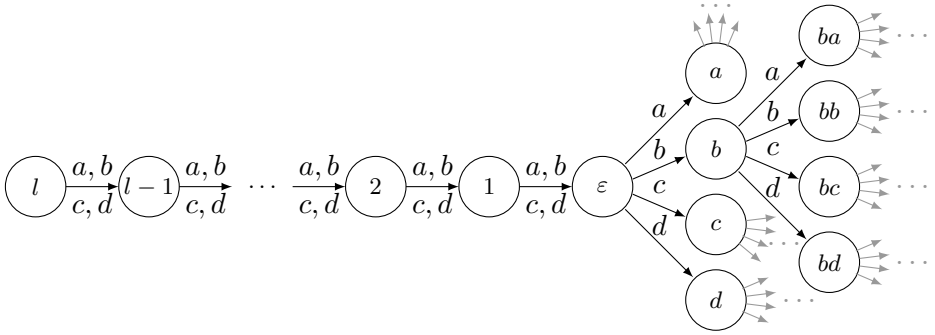

Figure 1: Representation of a l-countdown-append object as an automaton. The unrepresented read operation returns the label of the state on which it is applied.

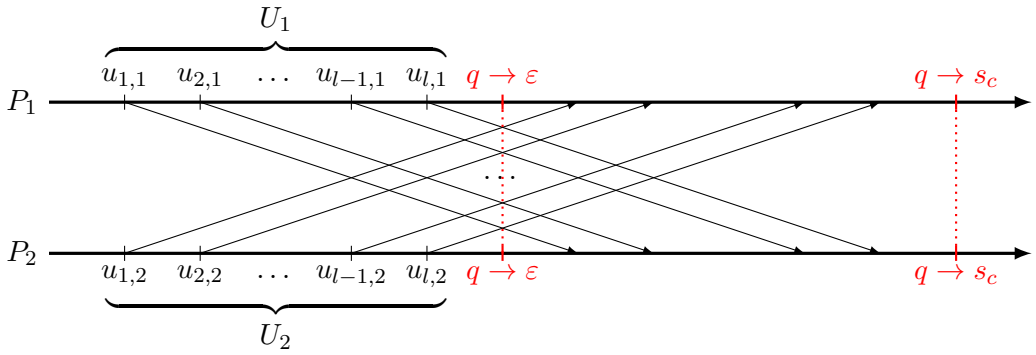

Figure 2: Typical execution in the proof of Theorem 1

5.2 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 [14]: 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.

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

Proof. Let A 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 (recall that causal deliveries implies FIFO order). 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, which 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 A is deterministic.

The executions can be distinguished 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 $\mathcal{X}_1(v_1) = \{X_{(v,v_2)} \in \mathcal{X}_1 : v = v_1\}$. As there are 2^l possible values for v_1, at least one of them is 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 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 A is at least $\frac{l}{2} - 1$ bits. □

5.3 Upper bound in the wait-free model

A consequence of Theorem 1 is that any update consistent universal construction in the operational model must maintain metadata whose size is at least linear in the number of update operations in some executions. Although this does not imply that this metadata must be stored in the form of a history containing all the update operations like in Algorithm 2, this does imply that the metadata required by UC_∞ cannot be drastically reduced, and in particular that it is impossible to safely prune the history of Algorithm 2 from old operations in the operational model. Differently, Theorem 1 does not apply to the more general wait-free model. In this section, we propose Algorithm 3, called UC_0, whose metadata mainly consist of a vector clock of size $O(n \log(m))$ where n is the number of processes and m the total number of issued query operations.

The principle of UC_0 is to dissociate the safety and liveness parts of update consistency. On the

Algorithm 3: Universal construction $UC_0(\mathcal{O}, \mathcal{R}, \mathcal{S}, s_0, u, q)$: code for p_i

```
1  operation apply(o ∈ O) ∈ R
2      var s ∈ S ← state_i;
3      if o is an update then broadcast mUpdate(o) ;
4      return q(s, o);
5  when a message mUpdate (o_j ∈ O) is received from p_j
6      state_i ← u(state_i, o_j); clock_i[j] ← clock_i[j] + 1; leader_i ← i;
7      broadcast mCorrect (clock_i, state_i);
8  when a message mCorrect (cl_j ∈ N^N, s_j ∈ S) is received from p_j
9      if clock_i = cl_j ∧ j ≤ leader_i then
10         state_i ← s_j; leader_i ← j;
```

safety side, each process maintains its own vision of the state of the object, making sure it results from some possible linearization of all the update operations it is aware of. This ensures that, if all processes but one crash, all the query operations performed by the remaining process are done in a consistent state. On the liveness side, processes exchange their visions of the state, striving to converge to a common state. More precisely, for a given set of update operations, represented by a version vector, processes will always adopt the state proposed by the process with the lowest identifier, which they call their leader. This ensures update consistency since, if all processes stop updating, all correct processes will eventually adopt the state forged by the correct process with the lowest identifier.

Process p_i maintains three variables. Variable $\mathsf{state}_i \in \mathcal{S}$ represents the current local state at p_i, initialized as the initial state s_0. Variable $\mathsf{clock}_i \in \mathbb{N}^{\mathbb{N}}$ is the equivalent of a version vector, such that $\mathsf{clock}_i[j]$ represents the number of update operations received by p_i from process p_j. As p_i does not know the number of participants, it is encoded as an associative array, rather than a vector. Finally, variable $\mathsf{leader}_i \in \mathbb{N}$ is the identifier of a process such that, if $\mathsf{clock}_i = \mathsf{clock}_{\mathsf{leader}_i}$, then p_i and p_{leader_i} are in the same local state.

When process p_i invokes a query operation, it issues it locally on its local state state_i (Line 4). When p_i invokes an update operation, it broadcasts a message `mUpdate` (Line 3). At reception of such a message from p_j, p_i executes the operation on its local state and updates its vector clock. It also resets its variable leader_i to i, which signifies that its local state was computed by itself. Then, p_i broadcasts a `mCorrect` message containing its current version of the state and the associated version vector. When p_i receives a `mCorrect` message from p_j, it first checks the version of the state by comparing the version cl_j of the message with its own version vector clock_i. Thanks to causal delivery, clock_i cannot be older than cl_j. If $cl_j[k] < \mathsf{clock}_i[k]$ for some k, the last updates of p_k are already known by p_i, but not by p_j. In that case, process p_i simply ignores the message. Otherwise, if $cl_j = \mathsf{clock}_i$, p_i keeps the state computed by the process with smallest identifier, which it remembers by updating its variable leader_i.

Theorem 2 (Correctness)**.** *Algorithm 3 is a wait-free, update consistent universal construction.*

Proof. The operation `apply` in Algorithm 3 terminates because it contains no loop.

Let H be 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. Let us suppose that H contains an infinite number of queries and a finite number m of updates (we only consider updates for which a message `mUpdate` was sent). Let us denote by m_i the number of updates performed by p_i, such that $m = \sum_{i=1}^{n} m_i$.

Some processes are correct because at least one of them performs an infinite number of queries, and all correct processes p_i send a message $\texttt{mCorrect}(cl_{max}, s_i)$, with $cl_{max} = [m_1, \ldots, m_n]$, after they have received all `mUpdate` messages (Line 7). Let p_{leader} be the process with the smallest identifier that sent such a message, called m_{leader}. Thanks to causal delivery, any correct process p_i received all `mUpdate` messages before m_{leader}, so at reception of m_{leader}, it had $\mathsf{clock}_i = cl_{max}$ and, by definition of p_{leader}, $leader < \mathsf{leader}_i$. Therefore, p_i adopted s_{leader} as its own state, and for any subsequent reception of a message $\texttt{mCorrect}(cl_j, s_j)$ by p_i, either $cl_j < \mathsf{clock}_i = cl_{max}$ or $leader = \mathsf{leader}_i < j$. Therefore, all correct processes p_i converge to the same state s_{leader}.

Let us now prove that, at any time, $\mathsf{clock}_i[j]$ represents the number of `mUpdate` received by p_i from p_j, and the state state_i of p_i can be obtained by a linearization of the corresponding updates. We prove this by induction on the number of times state_i and clock_i has changed at some process p_i. Initially, $\mathsf{clock}_i[j] = 0$ and $\mathsf{state}_i = s_0$. Suppose the property was always true until process p_i updates $\mathsf{clock}_i[j]$ or state_i. If this happens after reception of a `mUpdate` message from p_j, the property remains true because causal broadcast implies FIFO reception. If this happens after reception of a `mCorrect` message from p_j, then the property was previously true on process p_j, on the clock cl_j and the state s_j, so it remains true after the update on the clock $\mathsf{clock}_i = cl_j$ and the state $\mathsf{state}_i = s_j$.

Finally, all the query operations done after all messages have been received are done in the same state s_{leader}, that can be obtained by a linearization of all the update operations. This implies update consistency. □

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

Proof. Let $l \in \mathbb{N}$ and let O_l be the l-countdown-append object. Let $v \in U^l$. In any execution of Algorithm 3 abstracted by H_v, there is a process p_i that performs all l update operations. At the end of the execution, clock_j only contains one pair (i, l) that can be encoded in $O(\log(nl))$ bits, $\mathsf{state}_j = \varepsilon$ has a constant size, and leader_j is the identifier of a process, of size $O(\log(n))$ bits. Therefore, 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 A_{om} implementing an update consistent O in the operational model. By Theorem 1, there exists v such that the H_v-complexity of A_{om} is at least $\frac{l}{2} - 1$ bits. Therefore, Algorithm 3 has a strictly lower H_v-complexity than A_{om}. □

6 A practical solution

In previous sections, we encountered two different strategies in order to implement an update consistent universal construction. On the one hand, Algorithm UC_∞ belongs to the operational model, and therefore has an optimal complexity in the number of exchanged messages. The limit of this algorithm is that it requires an unbounded amount of memory and computation power, as the stored history may grow forever. According to Theorem 1, this limit cannot be overtaken while remaining in the operational model. On the other hand, Algorithm UC_0 manages to dispose of the stored history thanks to a synchronization mechanism that is forbidden in the operational model. The limitation of this algorithm is its complexity in network communication, as a complete state is broadcast by all processes for every updates.

This section presents Algorithm $UC[k]$, which takes the best of both methods. It builds on the observation that, in real systems, asynchrony is often used as a convenient abstraction for systems in which transmission delays are actually bounded, but the bound is too large to be used in practice, or unknown. This means that after some time, old messages from the first strategy can be garbage collected. The second strategy is used to ensure safety of this garbage collection: if old messages are received later than expected, the *a priori* total order can be altered and states must be exchanged to ensure convergence. Thus, the overhead in memory and computation time remains bounded by a parameter k that affects the size of the list, while more bandwidth is required only when messages are abnormally delayed, for example to recover from a partition. UC_∞ is the limit of $UC[k]$ when $k \to \infty$, and $UC[0]$ corresponds to an improvement of UC_0. The value k can be seen as a parameter that combines the issuing rate of update operations and the the message transmission delay.

6.1 Presentation of the universal construction

The code for $UC[k]$ executed by process p_i is given on Algorithm 4. The current local state of the object at process p_i is divided into two parts. On the one hand, the most recent updates known by p_i are stored with timestamps in a list of updates, in a similar fashion as in Algorithm UC_∞. On the other hand, older updates are pruned and recorded in a state of the object, that is maintained similarly to Algorithm UC_0. The separation between old and recent updates is arbitrary and abstracted by the parameter k of the algorithm: at any time, the difference between all timestamps in the list of recent updates must be at most k. Unlike in Algorithm UC_0, the local state of the object is not sent each time an update is received, but only after a conflict has been detected: a conflict occurs at p_i when p_i receives a `mUpdate` message for an update operation, whose timestamp indicates that it should already be part of the recorded state. The counterpart of this optimization is that a `mCorrect` message may be sent by p_i in response to another `mCorrect` message sent by p_j, if p_j detected a conflict but p_i did not. Each process p_i manages the seven following local variables. Variables $\mathsf{time}_i \in \mathbb{N}$, initially 0, and $\mathsf{history}_i \subset \mathbb{N} \times \mathbb{N} \times \mathcal{O}$, initially empty, are respectively the state of the Lamport clock and the ordered list of recent updates. They play a similar role as in Algorithm UC_∞, except that $\mathsf{history}_i$ is regularly pruned of its oldest updates. Variables $\mathsf{state}_i \in \mathcal{S}$, initially s_0, $\mathsf{clock}_i \in \mathbb{N}^{\mathbb{N}}$, initially $[0, \ldots, 0]$, and $\mathsf{leader}_i \in \mathbb{N}$, initially i, play a similar role as in Algorithm UC_0, but only encode older updates: state_i is the recorded state computed from old updates, and clock_i is the version vector that describes the set of updates used

Algorithm 4: Universal construction $UC[k](\mathcal{O}, \mathcal{R}, \mathcal{S}, s_0, u, q)$: code for p_i

```
1  operation apply(o ∈ O) ∈ R
2      var s ∈ S ← state_i;
3      if o is a query then
4          for (t, j, o') ∈ history_i sorted according to (t, j) do s ← u(s, o') ;
5      if o is an update then time_i ← time_i + 1; broadcast mUpdate (time_i, o) ;
6      return q(s, o);
7  function record(t ∈ N)
8      rtime_i ← max(rtime_i, t);
9      for (t_j, j, o_j) ∈ history_i with t_j ≤ rtime_i sorted according to (t_j, j) do
10         state_i ← u(state_i, o_j); history_i ← history_i \ {(t_j, j, o_j)};
11         clock_i[j] ← clock_i[j] + 1; leader_i ← i; sent_i ← false;
12 when a message mUpdate(t_j ∈ N, o_j ∈ O) is received from p_j
13     time_i ← max(time_i, t_j);
14     history_i ← history_i ∪ {(t_j, j, o_j)};
15     var conflict ∈ {true, false} ← t_j ≤ rtime_i;
16     record(time_i − k);
17     if conflict then sent_i ← true; broadcast mCorrect(clock_i, rtime_i, state_i) ;
18 when a message mCorrect(cl_j ∈ N^N, t_j ∈ N, s_j ∈ S) is received from p_j
19     record(t_j);
20     if clock_i = cl_j ∧ j < leader_i then
21         state_i ← s_j; leader_i ← j; sent_i ← true;
22     else if ¬sent_i then
23         sent_i ← true; broadcast mCorrect (clock_i, rtime_i, state_i);
```

by p_{leader_i} to craft state_i. Variable $\mathsf{rtime}_i \in \mathbb{N}$, initially $-k$, encodes the timestamp that separates old updates and recent updates. Usually, $\mathsf{rtime}_i = \mathsf{time}_i - k$, but it can be different if processes do not agree on the value of k, as discussed in Section 6.3. Finally, sent_i is a Boolean value, initially **true**, that encodes whether or not state_i was broadcast on the network, in order to minimize the traffic.

When process p_i executes a query operation, it first applies all the updates stored in $\mathsf{history}_i$ on state_i according to the lexicographic order on their timestamps and computes the return value by applying the query to the obtained state.

When process p_i executes an update operation o, it causally broadcasts a $\texttt{mUpdate}(t, o)$ message, where o identifies the update operation, and t is the virtual time generated using the Lamport clock time_i used in the timestamp of the operation.

When process p_i receives a message $\texttt{mUpdate}(t_j, o_j)$ from p_j, it first updates its local virtual time time_i and inserts o_j in the update list $\mathsf{history}_i$. Then, p_i applies all the updates whose timestamps are lower than $\mathsf{time}_i - k$ to state_i by executing $\texttt{record}(\mathsf{time}_i - k)$. Two cases are possible, depending on the relative values of t_j and of rtime_i at reception of the message. Normally, $\mathsf{rtime}_i < t_j$, which means that o_j is a recent update that belongs to $\mathsf{history}_i$, and p_i has nothing else to do. Otherwise, o_j should already have been included into state_i. By executing o_j on state_i, p_i may have jeopardized convergence since another process may have executed the operations in a different order. As p_i applied the update to state_i, it has changed the linearization order, but this new order is correct thanks to causal reception. It therefore notifies the other processes of its choice by broadcasting a message $\texttt{mCorrect}$ containing its new state.

When process p_i receives a message $\texttt{mCorrect}(cl_j, t_j, s_j)$ from p_j, it knows that p_j has faced a conflict and changed its linearization order. It can either accept or reject the correction, depending on the version vector cl_j associated to the state s_j of the correction, and the version vector clock_i associated to its own state state_i. Thanks to causal delivery, p_i has received at least the same messages $\texttt{mUpdate}$ before the message $\texttt{mCorrect}$, as p_j received before sending the message $\texttt{mCorrect}$. Process p_i then executes $\texttt{record}(t_j)$ to make sure that $\mathsf{clock}_i \geq cl_j$ (this is not necessary if all processes share the same value for k. The two following cases are possible:

- If $\mathsf{clock}_i = cl_j$, both states have been produced with the same updates but possibly in a different order. In this case, arbitration is done considering which process has the smallest identifier. If $j < \mathsf{leader}_i$ then p_i accepts the correction and update its local variables. Otherwise, p_i chooses to keep its own state. Instead, it broadcasts it so everyone can also adopt its state. If this situation occurs several times successively, it is necessary to compare all identifiers together, so the comparison is done with variable leader_i, that contains the identifier of the process whose state was chosen. The variable sent_i is used to prevent p_i from broadcasting the same state several times.
- If $\mathsf{clock}_i > cl_j$ then state_i takes updates into account which s_j does not. The future reception of these updates by p_j will lead to a new conflict, so p_i sends another $\texttt{mCorrect}$ message to help p_j solve the conflict.

6.2 Correctness

In this section, we prove that Algorithm 4 implements update consistency, i.e. that all the histories it allows are update consistent. To this end, we prove three intermediate lemmas: Lemma 1 ensures, among other properties concerning the global state of the system, that the virtual local state of a process is always valid with respect to the updates it has heard of; Lemma 2 proves that, if all processes stop updating, then eventually, no message will be sent to maintain consistency and in this situation, Lemma 3 proves that, all processes have converged to a common state. Finally, Theorem 3 proves that Algorithm 4 is update consistent. Remark that, whenever an operation is issued, it is completed without waiting for any other process, so the algorithm is wait-free.

We consider an object $O = (\mathcal{O}, \mathcal{R}, \mathcal{S}, s_0, u, q)$ and a history H that models an execution of Algorithm 4. Let us introduce the following notations.

- The superscript notation on variables of the algorithm, e.g. state_i^x, denotes the value of a variable at a time x.
- $vs_i^x \in \mathcal{S}$ denotes the virtual state of process p_i at time x, obtained by executing the operations contained in $\mathsf{history}_i^x$ on state_i^x, respecting the lexicographic order on their timestamps, similarly to the state on which the queries are done and contained in variable s after line 4.
- $vcl_i^x \in \mathbb{N}^{\mathbb{N}}$ denotes the virtual clock of process p_i at time x, that represents the set of `mUpdate` messages received by p_i at time x. It is defined, for all $j \in \{1, \ldots, n\}$, by $vcl_i^x[j] = \mathsf{clock}_i^x[j] + |\{(t, k, o) \in history_i^x : k = j\}|$.
- For $cl \in \mathbb{N}^{\mathbb{N}}$ and $s \in \mathcal{S}$, $cl \triangleright s$ expresses the fact that the prefix of the history containing the $cl[j]$ first update operations of each process p_j can lead to state s.

Lemma 1 (Validity)**.** *At each time x, for each process p_i that is not executing a part of the algorithm, and for each message* `mCorrect`(cl_i, t_i, s_i) *sent by p_i before time x, we have:*

1. $vcl_i^x \triangleright vs_i^t$,
2. $\mathsf{clock}_i^x \triangleright \mathsf{state}_i^x$,
3. $cl_i \triangleright s_i$.

Proof. We proceed by induction on the succession of the global states of the system. Initially, there is no message in transit, and for each process p_i, $[0, ..., 0] = \mathsf{clock}_i^0 = vcl_i^0 \triangleright vs_i^0 = \mathsf{state}_i^0 = s_0$. We suppose now that at a time x^-, the property is verified and the global state changes. We prove that the property is still verified at x^+, just after the changes, that we consider to happen on p_i. The variables mentioned in the lemma are only changed at the reception of a message, so two cases must be distinguished.

Reception of `mUpdate`(t_j, o_j) **sent by** p_j**.** Before calling the function `record`, $vcl_i^x[j]$ was incremented and vs_i^x was updated by the insertion of o_j in $\mathsf{history}_i$. As causal broadcast ensures the FIFO reception of messages from p_j by p_i, (1) still holds, while (2) and (3) remain unchanged.

During each iteration of the loop of `record`, neither vcl_i^x nor vs_i^x is modified, because the order in which the updates are sorted on line 9 and in the definition of vs_i^x are the same, so (1) remains valid. After the loop, (2) is still true since the updates are applied and remove from $\mathsf{history}_i$ in a order defined by a Lamport clock, which contains the process order.

The messages `mCorrect` present in the system are those present at t^-, which verify (3), and possibly the one sent on line 17, with $cl_i = \mathsf{clock}_i^{x+} \triangleright \mathsf{state}_i^{x+} = s_i$ so (3) holds.

Reception of $\texttt{mCorrect}(cl_j, t_j, s_j)$ **sent by** p_j**.** Like above, the properties are respected after the call to `record` (time x_R).

If the condition of Line 20 is true, (3) holds because no new message $\texttt{mCorrect}(cl_i, t_i, s_i)$ is sent by p_i and (2) holds because $\mathsf{clock}_i^{x^+} = cl_j \triangleright s_j = \mathsf{state}_i^{x^+}$. The virtual clock $vcl_i^{x^+}$ remains unchanged and the concatenation of the linearization given by (2), and all the updates in $\mathsf{history}_i^{x^+}$ is still a valid linearization that leads to $vs_i^{x^+}$ thanks to causal delivery, so (1) holds.

If the condition is false, (1) and (2) are not impacted, and (3) holds because the message $\texttt{mCorrect}(cl_i, t_i, s_i)$ sent by p_i is such that $cl_i = \mathsf{clock}_i^{x+} \triangleright \mathsf{state}_i^{x+} = s_i$.

□

Lemma 2 (Quiescence)**.** *If each process p_i performs a finite number m_i of updates, there is a time x_{quiet} such that, for all $x > x_{\text{quiet}}$, no message is in transit in the network at x, and for all processes p_i, $vcl_i^x = [m_1, ..., m_n]$.*

Proof. As messages `mUpdate` are only broadcast on line 5, when a new update is performed, only a finite number of them are sent. Moreover, p_i cannot send two messages `mCorrect` with the same value of clock_i: if the broadcast is done on line 17, *conflict* is true so clock_i was incremented just before during the call of `record` (clock_i is increased because, at least, the received update is applied since there was a conflict), and if the broadcast is done on line 23, sent_i was set to false in function `record` just after an increment of clock_i. As clock_i is bounded by $[m_1, ..., m_n]$, a bounded number of messages `mCorrect` are sent. As all the messages eventually arrive, there is a time x_{quiet} such that for all $x > x_{\text{quiet}}$, all the messages have been received and processed. In particular, all messages `mUpdate` have been received by p_i, so $vcl_i^{x_{\text{quiet}}} = [m_1, .., m_n]$. □

Lemma 3 (Convergence)**.** *At each time x, if there is no message in transit at x, then for all processes p_i and p_j, $vs_i^x = vs_j^x$.*

Proof. Let x be a time at which no message is in transit, and let p_i and p_j be two processes.

Let us first suppose that a conflict occurred during the execution, and that a message `mCorrect` was sent. Let us consider a message m_k of the form $\texttt{mCorrect}(cl_k, t_k, s_k)$ sent by p_k such that there is no other message $\texttt{mCorrect}(cl_l, t_l, s_l)$ sent by p_l with $cl_l > cl_k$, or $cl_l = cl_k$ and $l < k$ (m_k might not be unique since the order on version vectors is partial, but it exists).

As no message is in transit at time x, p_i already received m_k, and after the execution of `record` at time x_i, we have $\mathsf{clock}_i^{x_i} \geq cl_k$ thanks to causal delivery. We know that $\mathsf{clock}_i^{x_i} = cl_k$ and $k < \mathsf{leader}_i^{x_i}$ since, otherwise, $p_{\mathsf{leader}_i^{x_i}}$ would have sent a message $m_i = \texttt{mCorrect}(\mathsf{clock}_i^{x_i}, t, \mathsf{state}_i^{x_i})$, either because p_i received it if $\mathsf{leader}_i^{x_i} \neq i$, or because p_i executed Line 23 if $\mathsf{sent}_i^{x_i}$ was false

or Line 23 if $\mathsf{sent}_i^{x_i}$ was true. In all cases, m_i contradicts the definition of m_k so it cannot exist. Therefore, p_i executed Line 21, and accepted the correction. Moreover, no further conflict occurred at p_i because such a conflict would result in a message `mCorrect` associated to a version vector strictly larger than cl_k, contradicting the definition of m_k. Similarly, p_j adopted the correction of m_k at some time x_j and was not subject to any conflict later.

If no conflict occurred during the execution, let us pose $x_i = x_j = 0$ the initial time. In both cases, we have $\mathsf{clock}_i^{x_i} = \mathsf{clock}_j^{x_j}$, $\mathsf{state}_i^{x_i} = \mathsf{state}_j^{x_j}$, and no conflict occurred on p_i after x_i, nor on p_j after x_j. Then, every message $\texttt{mUpdate}(t_k, o_k)$ received by p_i at a time $x_r > x_i$ is associated to a virtual time $t_k > \mathsf{rtime}_i^{x_r}$, and the same can be said about p_j. Therefore, p_i and p_j apply all the updates in `record` according to the same lexicographic order, which leads them to the same virtual state $vs_i^x = vs_j^x$. □

Theorem 3 (Correctness)**.** *All histories allowed by Algorithm 4 are update consistent.*

Proof. Let H be a history allowed by Algorithm 4. If H contains an infinite number of updates or a finite number of queries, it is update consistent by definition. Let us suppose that each process p_i performs a finite number m_i of update operations. According to Lemma 2, there is a time x_{quiet} such that, for all $x > x_{\text{quiet}}$, no message is in transit in the network at x, and for all i, $vcl_i^x = [m_1, ..., m_n]$. By Lemma 3, there is a state $s_{\text{conv}} \in \mathcal{S}$ such that, for all process p_i and all $x > x_{\text{quiet}}$, $vs_i^x = s_{\text{conv}}$. Moreover, as $vcl_i^x = [m_1, ..., m_n]$ and according to Lemma 1.1, $vcl_i^x \triangleright vs_i^x$, which means there is a linearization of all the updates that leads to this common state s_{conv}, in which all the queries done after x_{quiet} are done. As only a finite number of queries were done before x_{quiet}, H is update consistent. □

6.3 The trade-off k

There are two ways to understand Algorithm $UC[k]$. On the one hand, it can be seen as a garbage collection mechanism added to Algorithm UC_∞, that allows the safe disposal of old operations. In that sense, the smaller the k, the least memory is necessary. On the other hand, Algorithm $UC[k]$ can be seen as an optimization of Algorithm UC_0 in which the list of updates $\mathsf{history}_i$ is used as a cache to store new updates as long as there is a risk that concurrent updates may be received, and thereby reducing the need for broadcasting correction messages. In that sense, the larger the k, the least frequent "cache-misses" should occur, and the least communication-intensive the algorithm is. Thus, the value k is a trade-off between the space used and the number of sent messages.

For each execution, there exists, however, a minimal value of k for which no `mCorrect` message is sent. Any larger value for k increases the memory needs while keeping the communication complexity constant, and any smaller value for k generates conflicts that cause the broadcast of expensive `mCorrect` messages. Since the timestamps of the list of updates $\mathsf{history}_i$, that span an interval of size at most k, are generated using a Lamport clock that encodes the "happened-before" relation, this optimal value corresponds to the length of the longest causal chain of update operations done during the transit of a single `mUpdate` message. This value depends on the transmission delay of messages, both relatively to other messages and relatively to the time between two updates done by a process.

The value of k has an important impact of the performances of the algorithm, but choosing the perfect value requires knowledge about the future execution. Interestingly, k is not used in the proof of correctness. An important consequence is that the algorithm remains correct even if processes do not share a common value of k. In particular, each process can update its value of k independently, depending on its own characteristics such as its available memory, or following some global strategy to adapt k to the specificities of the system. While updating a global parameter is highly costly, updating a local value for k is cheap: increasing k is free, while decreasing k only consists in executing $\texttt{record}(\mathsf{time}_i - k)$ with the new value of k.

7 Conclusion

This paper explores the complexity of update consistent universal constructions in message passing systems composed of asynchronous processes that may fail by crashing. To be able to order operations despite asynchrony, UC_∞, a previously-proposed construction, stores all operations in an ever-growing log. Although this construction is optimal in the number of exchanged messages, its space complexity may be prohibitive for many applications. The goal of this paper is to reduce the space complexity of update consistent universal constructions, but keeping the message complexity as low as possible.

Our first contribution is a proof that the optimal message pattern adopted in Algorithm UC_∞, called the operational model, makes it necessary to store $O(m)$ bits of information for an execution containing $O(m)$ operations, which is the same asymptotic complexity as Algorithm UC_∞. Contrastingly, we proved that algorithms with a logarithmic spatial complexity were available in the more general wait-free model, at the price of a greater communication complexity.

This is an important theoretical result, as it negatively answers the following question: are the wait-free model and the operational model equivalent in terms of space complexity? It 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.

Our second contribution is a new update consistent universal construction. The performance of the proposed algorithm depends on a parameter k which cannot be chosen perfectly without knowledge about the future execution. A too low value of k may make the execution costly in terms of communication, while a high value of k requires more memory and computation resources. A possible future work would be to explore the best strategies to adapt dynamically k in function of some communication conditions, to optimize the need for resources.

Acknowledgments

This work was partially supported by the French ANR project 16-CE25-0005 O'Browser.

References

[1] R. J. Lipton, J. S. Sandberg, PRAM: A scalable shared memory, Princeton University, Department of Computer Science, 1988.

[2] H. Attiya, J. L. Welch, Sequential consistency versus linearizability, ACM Transactions on Computer Systems (TOCS) 12 (2) (1994) 91–122.

[3] H. Attiya, A. Bar-Noy, D. Dolev, Sharing memory robustly in message-passing systems, J. ACM 42 (1) (1995) 124–142.

[4] W. Vogels, Eventually consistent, Queue 6 (6) (2008) 14–19.

[5] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, P. W. Hutto, Causal memory: Definitions, implementation, and programming, Distributed Computing 9 (1) (1995) 37–49.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, W. Vogels, Dynamo: amazon's highly available key-value store, in: ACM SIGOPS Operating Systems Review, Vol. 41, ACM, 2007, pp. 205–220.

[7] M. Shapiro, N. Preguiça, C. Baquero, M. Zawirski, Conflict-free replicated data types, in: Symposium on Self-Stabilizing Systems, Springer, 2011, pp. 386–400.

[8] M. Perrin, A. Mostefaoui, C. Jard, Update consistency for wait-free concurrent objects, in: International Parallel and Distributed Processing Symposium, IEEE, 2015, pp. 219–228.

[9] M. Perrin, A. Mostefaoui, C. Jard, Causal consistency: Beyond memory, in: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'16), ACM, 2016, p. 26.

[10] M. Raynal, A. Schiper, S. Toueg, The causal ordering abstraction and a simple way to implement it, Information processing letters 39 (6) (1991) 343–350.

[11] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Communications of the ACM 21 (7) (1978) 558–565.

[12] B. Randell, P. Lee, P. C. Treleaven, Reliability issues in computing system design, ACM Computing Surveys (CSUR) 10 (2) (1978) 123–165.

[13] R. Baldoni, J. Brzezinski, J.-M. Hélary, A. Mostefaoui, M. Raynal, Characterization of consistent global checkpoints in large-scale distributed systems, in: Workshop on Future Trends of Dist. Computing Systems, IEEE, 1995, pp. 314–323.

[14] S. Burckhardt, A. Gotsman, H. Yang, M. Zawirski, Replicated data types: specification, verification, optimality, in: ACM Sigplan Notices, Vol. 49, ACM, 2014, pp. 271–284.

[15] H. Attiya, F. Ellen, A. Morrison, Limitations of highly-available eventually-consistent data stores, IEEE Trans. Parallel Distrib. Syst. 28 (1) (2017) 141–155.

[16] H. Attiya, S. Burckhardt, A. Gotsman, A. Morrison, H. Yang, M. Zawirski, Specification and complexity of collaborative text editing, in: Symposium on Principles of Distributed Computing, ACM, 2016, pp. 259–268.