Does the operational model capture partition tolerance in distributed systems?

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

1 Introduction

Eventual consistency. In large scale distributed systems, replication is essential in order to provide availability and partition tolerance. 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 [6] 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.

The operational model. In this context, Conflict-Free Replicated Data Types (CRDTs) [11] 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. Thanks to commutativity, 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 is more problematic and has been widely studied for several objects including sets, counters and registers [5], data stores [2] and collaborative editors [1].

The wait-free model. Despite the fact that algorithms from the operational model are naturally partition tolerant and minimize communication 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 [9, 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: 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*. Failure tolerance 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.

Processes can communicate by sending and receiving messages, using the *causal broadcast* abstraction[1] that provides them with a **broadcast**(m) operation and a **receive**(m) event, where m is a message. Communication channels are uniformly reliable, as all correct processes eventually receive the same set of messages, including their owns. However, channels are asynchronous, in the sense that there is no bound on the time it takes for one message to be delivered.

A history in the wait-free model is an abstraction of an execution that contains the information accessible for an outside observer, i.e. the operations that were performed, their invoking process and time, as well as their returned value.

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).

More precisely, given a history H that contains a finite number of updates, and an algorithm Λ, we define the H-complexity of Λ as follows. Let S be the

[1] Note that causal broadcast can be easily implemented in the wait-free model [10]. 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 of the complexity results we obtain in the next sections.

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 $|S| = \infty$ (i.e. if S has states of unbounded size), the H-complexity is ∞;
- otherwise, the H-complexity is the maximal size of a state in S.

Problem statement. The wait-free model is strictly more general than the operational model, as any algorithm from the operational model is also 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 propose a new object, called *update consistent l-countdown-append object*, and compare its wait-free implementations in both models. As its name suggests, the update consistent l-countdown-append object is specified by a sequential specification, that describes the behaviour of the object when processes access it sequentially, and a weak consistency criterion, called update consistency [8], that describe how concurrency affects the sequential behaviour of the object.

The l-countdown-append object, where $l \in \mathbb{N}$, exposes the 4 update operations in the set $U = \{a, b, c, d\}$, and one query operation, q. The behaviour of the object 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.

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. More formally, a history H is update consistent for an object O if it is in one of the two following cases:

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

On a computability viewpoint, it is possible to implement any object with this criterion in both computing models [8].

Contributions. This paper proves that the two models are not equivalent: we prove that $\mathcal{O}(l)$ bits are necessary in the operational model to implement an update consistent l-countdown-append, but give a logarithmic algorithm in the wait-free model.

Organization. Section 2 proves the part of the result for the operational model, and Section 3 explores the wait-free model. Finally, 4 concludes the paper. We could not include all the proofs in this extended abstract, due to space restrictions. A complete version of the paper can be found in [4].

2 Lower bound in the operational model

In order to compare both models, we introduce a class of histories: the H_v histories. Let $l \in \mathbb{N}$, and $v = u_1...u_l$ be a word consisting of l update operations of the l-countdown-append object. We denote by H_v the history in which one process performs all updates of v in their order of appearance, and the other processes keep performing the query operation.

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 [5]: 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 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 $X_{(v_1,v_2)}$ 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.

3 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 1, based on the algorithm UQ_0 from [7].

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 $\mathsf{clock}_i(j)$ 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, if $\mathsf{clock}_i < \mathsf{clock}_{\mathsf{leader}_i}$ 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 `mUpdate` (Line 8). 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 `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.

```
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 else return countdown_i;
7  operation u() // u ∈ U
8      broadcast mUpdate (clock_i[i] + 1, i, u);
9  receive mUpdate (t_j ∈ ℕ, j ∈ ℕ, u ∈ U)
10     if clock_i[j] < t_j then
11         clock_i[j] ← t_j; leader_i ← i;
12         if countdown_i = 0 then
13             append_i ← append_i · u;
14             broadcast mCorrect (clock_i, i, append_i);
15         else countdown_i ← countdown_i − 1;
16 receive mCorrect (cl_j ∈ Array(ℕ, ℕ), j ∈ ℕ, a_j ∈ U*)
17     if (∀k, clock_i[k] ≤ cl_j[k]) ∧ (j ≤ leader_i ∨ ∃k, clock_i[k] < cl_j[k]) then
18         append_i ← a_j; clock_i ← cl_j; leader_i ← j;
```

Algorithm 1: The countdown-append object in the wait-free model

Algorithm 1 is clearly wait-free as its operations contain no loop. It is also update consistent because, 1) all processes constantly maintain a state obtained by a linearization of the operations of their causal past, and 2) after all updates have been performed, all replicas converge to a common state, that is the state of the correct process with the smallest identifier.

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. For all processes p_j, clock_j only contains one entry for p_i, smaller than l. Therefore, clock_j can be encoded in 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 and $\mathsf{append}_i = \varepsilon$ is a constant value, so it has an encoding of constant size c. Finally, the H_v complexity of Algorithm 1 is $\mathcal{O}(\log(nl))$ bits, which proves the following theorem.

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.*

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}.*

4 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 when implemented on each of 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 $\mathcal{O}(n \log(nl))$ 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.

These two results show 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. Bonin, G., Achour, M., Perrin, M.: Does the operational model capture partition tolerance in distributed systems? extended version
5. 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)
6. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. Acm Sigact News (2002)
7. Perrin, M.: Distributed Systems: Concurrency and Consistency. Elsevier (2017)
8. 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)
9. Randell, B., Lee, P., Treleaven, P.C.: Reliability issues in computing system design. ACM Computing Surveys (CSUR) **10**(2), 123–165 (1978)
10. 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)
11. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In: Symposium on Self-Stabilizing Systems. pp. 386–400. Springer (2011)