Validated Objects: Specification, Implementation, and Applications

Antonio Fernández Anta
IMDEA Networks Institute, Spain

Chryssis Georgiou
University of Cyprus, Cyprus

Nicolas Nicolaou
Algolysis Ltd, Cyprus

Antonio Russo
IMDEA Networks Institute, Spain

Abstract

Guaranteeing the validity of concurrent operations on distributed objects is a key property for ensuring reliability and consistency in distributed systems. Usually, the methods for validating these operations, if present, are wired in the object implementation. In this work, we formalize the notion of a *validated object*, decoupling the object operations and properties from the validation procedure. We consider two types of objects, satisfying different levels of consistency: the validated *totally-ordered* object, offering a total ordering of its operations, and its weaker variant, the validated *regular* object. We provide conditions under which it is possible to implement these objects. In particular, we show that crash-tolerant implementations of validated regular objects are always possible in an asynchronous system with a majority of correct processes. However, for validated totally-ordered objects, consensus is always required if a property of the object we introduce in this work, *persistent validity,* does not hold. Persistent validity combined with another new property, *persistent execution*, allows consensus-free crash-tolerant implementations of validated totally-ordered objects. We demonstrate the utility of validated objects by considering several applications conforming to our formalism.

2012 ACM Subject Classification Computing methodologies → Distributed algorithms

Keywords and phrases Validation, Concurrent objects, Fault-tolerance, Distributed computing.

Funding *Antonio Fernández Anta*: Partially supported by Spanish State Research Agency (AEI) ECID project PID2019-109805RB-I00, cofunded by FEDER.

1 Introduction

Motivation. In distributed computing research, there is an important line of work on the formalization and implementation of distributed concurrent objects. A fundamental challenge of these implementations is making sure the operations that modify the state of an object never drive it into an incorrect or inconsistent state. In most proposals, the operations (and their arguments) invoked on the object have been assumed to be always valid, or ensuring this validity has been delegated to the application layer. With the popularization of public data structures (due to the wide usage and vast application scope of distributed ledger technologies), there is a growing interest on algorithms and objects capable of tolerating non-compliant user behavior. In this context, the implementation of an object cannot assume anymore that the operations invoked in the object will be well formed and respect any specification rule. Hence, the implementation of the object must be cautious, and validate operations before applying them. The direct way to do this is to introduce validation tests into the code that implements the object, so that an invalid operation execution is interrupted before it damages the object's state.

In this paper we explore the possibility of separating an object's implementation from the validation of the operations invoked in the object, and the implications of this separation. This approach is inspired by *aspect-oriented programming* [19], which aims in modular programming by separating cross-cutting concerns, i.e., cohesive areas of functionality. The idea is to add specific checks (advices as called) without changing the code of a program (object in our case). Our work is meant to be a first step on understanding how the application requirements and properties impact the algorithms that implement a distributed object through the introduction of a validation predicate valid() that wraps the application logic of the object.

Our approach and contributions. We employ a modular approach in which the characteristics and methods to validate the operations of an object are not "wired" in the object implementation. In particular, given a concurrent object O and its supported set of operations OP, we recast this object as a *validated* object via an apply() function. This function includes a validation filter, so that a specific operation $op \in OP$ is validated before it is executed, in accordance to an associated validation predicate valid(). Different validation predicates can be enforced via the apply() function without affecting the core code of the object.

Consider the following example. Let O be a simple R/W register supported by two operations, read(), which returns the value of the register, and write(v), which changes the value of the register into v. Say that we would like to impose that only positive numbers are written on the register. One approach would be to include a test directly in the code of the write function (see Code 1). However, should a different or an additional rule be needed, the code would have to be changed again, possibly jeopardizing the implementation correctness (especially in the the case of complex objects).

Code 1 Implementation of a positive R/W register O.

```
val ← ⊥
function read( )
    return (val)
function write(v)
    if v > 0 then
        val ← v
        return (ACK)
    else return (NACK)
```

With our approach, we separate the test from the code implementing the object. Processes invoke the desired operation via an apply() function. The process passes to apply() the same parameters as it would do in the "normal" case, and the apply function invokes a valid() predicate that has incorporated the desired validation test (i.e., in the case of a write operation, that v is positive, see Code 2). In case it is true, it then invokes execute(), which applies the operation on the object (i.e., it sets v as the value of the register). In case the validation fails (e.g., a negative value was intended to be written), apply() will return a NACK, signaling the violation of the imposed restriction (see Code 3). Should we require a different validation (e.g., we want a Boolean register), we would only replace the test in the valid() predicate (e.g., $v > 0$ in Line 3 becomes $v \in \{True, False\}$), without making any change on the object's implementation (i.e., in function execute()).

Code 2 Functions valid and execute for a positive R/W register O.

```
1: function valid(op)
2:     return (op = read()∨
3:             (op = write(v) ∧ v > 0))
4: function execute(op)
5:     if op = write(v) then
6:         val ← v; return (⊥)
7:     else return (val)
```

A particular challenge of our approach is to implement the validated version of a given object on a decentralized setting while guaranteeing certain level of consistency. In this work, we consider two types of validated objects, each providing a different level of consistency, the validated *regular* object and the *totally-ordered* one. Intuitively, a regular object provides consistency guarantees similar to a regular register [20], while the totally-ordered property is similar to linearizability [17]. We are now ready to summarize the contributions of this work.

- We introduce the formalization for a generic validated object O, along with the two mentioned consistency types, on which the application-specific operations are called (Section 2).

- We provide an algorithm to implement validated regular objects in crash-prone asynchronous distributed systems (Section 3).
- We provide an algorithm to implement validated totally-ordered objects under crash or Byzantine failures using the corresponding version of consensus [21, 3] (Section 4.1).
- Then in Section 4.2, we define a property of a validity predicate, which we call *persistent validity*, and in Section 4.3 we show that validated totally-ordered objects without persistent validity can be used to solve consensus.
- In Section 4.4, we introduce an additional property, that we call *persistent execution, which allows* a validated totally-ordered object to be implemented without consensus.
- Finally, in Section 5, we present some applications (such as a punching system and a cryptocurrency) that conform to the formalism we provided, demonstrating its usability.

Related work. The impact of a validation function has been already treated by previous work, according, however, to *specific* use cases.

In [12], a validity property called *forward acceptability* is defined, which enables the operations of a generic application to be commutative. This work only considers eventually consistent objects with this property. It provides an algorithm for a specific case, a PC-Ledger, that is implemented in a consensus-free system. On our side, we have a wider focus, including *any* object, characterizing its validity function and going in detail with the different consistency properties we are able to guarantee.

In [4], the authors introduce and solve the notion of Validated Byzantine Agreement to ensure that the decided value is one proposed by a non-faulty process. To do so, they enhance the system with an external validity condition, which requires that the agreement value is valid according to a global, polynomial-time computable predicate, known to all processes and it is application-determined. Hence, each process proposes a value that should satisfy this predicate. Such an external validity condition could be implemented using our approach via an appropriate apply() function and valid() predicate (which would implement the required predicate).

In blockchain systems, records are usually validated after the total order is globally agreed, validating and executing transaction according to the agreed sequence. As an example, Ethereum [26] first constructs a block, and then network nodes sequentially run the Ethereum Virtual Machine on each transaction to validate it, and update the global state if valid. This brings to the acceptance and inclusion in the block of invalid transaction inside the global order, in order to gain time in the consensus challenge of the system. A mitigation to this problem is brought by [8], that is build on top of [7], where validation is run by a subset of nodes before the proposal is broadcast to the whole network, in order to not overload nodes. In our work we abstract and generalize these behaviors, mapping them to validated objects with different consistency criteria.

In [1], the authors define a Validated Distributed Ledger Object. The validation is only taken into account in respect to Ledger Objects limiting the scope to that particular kind of data structure. Furthermore, the authors do not investigate or characterize the properties required by validation; they only assume the existence of a validation predicate.

2 Validated Objects

2.1 Concurrent Objects and Histories

We recall the general definition of object formalized in [1] where an object type T specifies (i) the set of values (or states) that any object O of type T can take, and (ii) the set of operations that a process can use to modify or access the value of O. An object O of type T is a concurrent object if it is a shared object accessed by multiple processes [24, 3]. Each operation on an object O consists of an invocation event and a response event, that must occur in this order. A history of operations on O, denoted by H_O, is a sequence of invocation and response events, starting with an invocation event. (The sequence order of a history reflects the real time ordering of the events.) An operation π is complete in a history H_O, if H_O contains both the invocation and the matching response of π, in this order. A history H_O is complete if it contains only complete operations; otherwise it is partial [24]. As in [24], we convert a partial history to a complete one by, for each incomplete operation π, either removing the invocation of π or completing π with a response event. From this point onward, we consider only complete histories. An operation π_1 precedes an operation π_2 (or π_2 succeeds π_1), denoted by $\pi_1 \rightarrow \pi_2$, in H_O, if the response event of π_1 appears before the invocation event of π_2 in H_O. Two operations are concurrent if none precedes the other. A *run* R of a distributed system that implements object O generates a (potentially infinite) history H_O.

2.2 Validated Object Types

In this work we consider validated objects. These are concurrent objects in which the operations executed and how they are interleaved are filtered with a predicate valid(). This predicate has as argument the state of the object S and a new operation op issued by process i, and it determines whether op is valid in the light of S. The state S is given by an ordered set of operations that have been executed in the object (and are valid). (The operations in S could be concurrent with op but have been "applied" in the object before it.) A second function execute() has the same arguments as valid() and, if the operation op is valid, is used to obtain the value that op returns. We will use the term *operation* and the symbol op for custom object logic unknown to our formalism, and we refer to *functions* when referring to the primitives of the objects we define, e.g., valid or execute.

Code 3 Centralized implementation of the apply function for a validated object O. Code executed by the central server. Function execute(S, op, i) provides the result of operation op by process i in state S. The operator $||$ combines the new operation with the previous valid executed operations.

$S \leftarrow \emptyset$ is the state of the object
function apply(op, i)
    if valid(S, op, i) **then**
        $r \leftarrow$ execute(S, op, i)
        $S \leftarrow S || op$
        return (ACK, r)
    else **return** $(NACK, -)$

In order to use a validated object, a client i invokes a function apply(op, i), which checks whether the operation op invoked by i is valid, and if so it applies it in the object by executing op. If the operation is not valid, the call apply(op, i) returns $(NACK, -)$. If the operation is valid, apply(op, i) returns (ACK, r), where r is the value that op returns. Code 3 shows a centralized implementation of the function apply(), executed at a single central server. This code is provided for illustration purposes. The operator $||$ that defines how the operations are combined into the state S is not detailed on purpose.

Observe that the history H_O of a run R of a validated object O contains *only* the operations op that are found valid and are in fact executed. These are the operations for which $\mathsf{apply}(op, i)$ returns (ACK, r). We denote the set of complete operations in history H_O generated in run R by $C(R)$.

In the following we assume that $\mathsf{valid}()$ and $\mathsf{execute}()$ have the following arguments:

- A strict partially ordered set of operations, given as a pair $\langle P, \prec\rangle$. P is a set of operations and $\prec$ is a strict partial order defined in P. In the especial case in which $\prec$ is a total order, denoted as $\lessdot$, the first argument can be provided as a sequence of operations.
- The operation op to be considered.
- The process i that issued op.

In this work we consider two types of validated objects.

▶ **Definition 1.** *A validated object specified with functions* $\mathsf{valid}()$ *and* $\mathsf{execute}()$ *is a* validated regular object *if in every run R a partial order $\prec$ among the set $C(R)$ of complete operations can be defined, such that,*

1. $\forall op, op' \in C(R), op \to op' \implies op \prec op'$;
2. $\forall op \in C(R)$, *let* $P(op) = \{op' : op' \in C(R) \land op' \prec op\}$ *and client i the issuer of op. Then,* $\mathsf{valid}(\langle P(op), \prec\rangle, op, i) = True$ *and op returns in its response event the value* $\mathsf{execute}(\langle P(op), \prec\rangle, op, i)$.

The following is a stronger version in which operations are totally ordered.

▶ **Definition 2.** *A validated object specified with functions* $\mathsf{valid}()$ *and* $\mathsf{execute}()$ *is a* validated totally-ordered object[1] *if in every run R a total order $\lessdot$ among the set $C(R)$ of complete operations can be defined, such that,*

1. $\forall op, op' \in C(R), op \to op' \implies op \lessdot op'$;
2. $\forall op \in C(R)$, *let* $P(op) = \{op' : op' \in C(R) \land op' \lessdot op\}$ *and client i the issuer of op. Then,* $\mathsf{valid}(\langle P(op), \lessdot\rangle, op, i) = True$ *and op returns in its response event the value* $\mathsf{execute}(\langle P(op), \lessdot\rangle, op, i)$.

In a run R of a validated totally-ordered object, the set $C(R)$ is totally ordered by $\lessdot$. We will denote the resulting *sequence* of all the operation of R by $S(R)$.

3 Algorithms Implementing Validated Regular Objects

We present algorithms implementing validated regular objects in an asynchronous system.

3.1 Model

We assume a distributed system composed of n processes with unique identities from the set $\mathcal{I} = \{1, \ldots, n\}$. Processes are asynchronous and crash prone, i.e., they advance their execution at arbitrary speed and can stop permanently (i.e., crash) at any point during their execution. Each process i has write access to a linearizable SWMR Distributed Ledger Object (DLO) [1, 5] denoted L_i. All processes can read all DLOs. A DLO L_i has a state, which is a totally ordered sequence S of records, initially empty, and has two operations:

- $L_i.get()$, which returns the current state (sequence of records) S of the DLO,
- $L_i.append(r)$, which adds record r to the end of the sequence S.

[1] Note that if $\mathsf{valid}()$ is considered as the sequential specification of the object, then the totally-ordered property is a form of linearizability defined over the executed operations. However, to avoid confusion, we prefer to use a different name since we do not include in the object histories the operations that were rejected with $(NACK, -)$ by $\mathsf{apply}()$.

These DLOs are reliable in the sense that any invocation to these operations by a correct process eventually completes [1]. Reliable linearizable SWMR DLOs can be implemented in an unreliable asynchronous system. This follows from the work of Imbs et al. [18], in which they implement SWMR atomic h-registers, which are registers that, when read, return the whole history of written values. Each of these h-registers trivially implements a SWMR DLO. Moreover, their implementation is on a distributed system with n servers, out of which up to $f < n/3$ can be Byzantine. Hence, the implementation is Byzantine-tolerant with optimal resilience. In this section we also observe that, with minor changes, the algorithm proposed by Imbs et al. [18] implements SWMR atomic h-registers for n crash-prone processes, out of which $f < n/2$ can fail.

As usual, we assume the following well-formedness property: A process i does not invoke a call to the function $\mathsf{apply}(op, i)$ of the object being implemented before the previous one has finished.

3.2 Crash-tolerant Algorithm for Validated Regular Objects

Let us consider a validated regular object O specified by the functions $\mathsf{valid}()$ and $\mathsf{execute}()$. Code 4 presents an implementation of the $\mathsf{apply}()$ function to be run by each of the processes of the distributed system in order to implement an instance of the object O. For technical reasons, we assume that the invocation action of a valid operation *op* issued by i occurs when it enters the loop in Line 2 and invokes $L_1.get()$, and that it completes when the $L_i.append(\langle ts, op\rangle)$ operation in Line 9 completes. What happens before and after these two actions respectively in the execution of $\mathsf{apply}(op, i)$ is local to process i, and not visible outside the client. This assumption removes the uncertainty of whether an operation has been completed if the process crashes after Line 9 but never executes Line 10.

Code 4 Crash-tolerant implementation of the apply function for a validated regular object O that uses linearizable SWMR DLOs $L_j, j \in [1, n]$. The code is for process $i \in [1, n]$.

```
 1: function APPLY(op, i)
 2:     for j = 1 to n do
 3:         G_j ← L_j.get()
 4:         T_j ← |G_j|
 5:     ts ← (i, T_1, ..., T_i, ..., T_n)
 6:     P ← {op' : ⟨ts', op'⟩ ∈ ⋃_j G_j}
 7:     if valid(⟨P, ≺⟩, op, i) then
 8:         res ← execute(⟨P, ≺⟩, op, i)
 9:         L_i.append(⟨ts, op⟩)
10:         return (ACK, res)
11:     else return (NACK, −)
```

We proceed by demonstrating that Code 4 implements a validated regular object as defined in Definition 1. Observe from Code 4 that every operation *op* issued by process i is assigned a timestamp $ts(op) = (i, T_1, \ldots, T_i, \ldots, T_n)$, which is appended as part of the record of *op* in ledger L_i if it is valid and completes. The value T_j in the timestamp is the number of records found in ledger L_j in the loop of Line 2. These timestamps are used to define the partial order $\prec$ among completed operations.

▶ **Definition 3.** *Given any two completed operations* $op, op' \in C(R)$*, with respective timestamps* $ts(op) = (i, T_1, \ldots, T_i, \ldots, T_n)$ *and* $ts(op') = (k, T'_1, \ldots, T'_i, \ldots, T'_n)$*,* $i, k \in [1, n]$*, then* $op \prec op'$ *if and only if* $T_i < T'_i$*.*

We show now that $\prec$ is a strict partial order as required.

▶ **Lemma 4.** *If* $op \prec op'$ *it cannot happen that* $op' \prec op$*. Hence,* $\prec$ *is a strict order.*

Proof. Assume for contradiction that $op \prec op'$ and $op' \prec op$. Let $ts(op) = (i, T_1, \ldots, T_n)$ and $ts(op') = (k, T'_1, \ldots, T'_n)$. Then $\mathsf{apply}(op, i)$ finds T_i records in ledger L_i and T_k records in ledger L_k, while $\mathsf{apply}(op', k)$ finds T'_i records in L_i and T'_k records in ledger L_k. By assumption, we have that $T_i < T'_i$ and $T_k > T'_k$. From $T_i < T'_i$ and the linearizability of L_i,

Code 5 Implementation of the *apply* function for a validated totally-ordered object O that uses an Atomic Broadcast service. The code is for process $i \in [1, n]$.

```
S ← ∅                         ▷ S is a sequence of operations
function APPLY(op, i)
    ret ← ⊥
    AB.broadcast(⟨op, i⟩)
    wait until ret ≠ ⊥
    return (ret)

upon (AB.deliver(⟨op, j⟩)) do
    if valid(S, op, j) then
        r ← execute(S, op, i)
        S ← S||op
        if j = i then ret ← (ACK, r)
    else
        if j = i then ret ← (NACK, −)
```

the append operation in $\mathsf{apply}(op, i)$ precedes or is concurrent with the $L_i.get()$ operation in $\mathsf{apply}(op', k)$. Hence, i executed $L_k.get()$ before k invoked $L_k.append(\langle ts(op'), op'\rangle)$. By the linearizability of L_k, it is not possible that $T_k > T'_k$, and we have a contradiction. ◀

▶ **Lemma 5.** $op \rightarrow op' \implies op \prec op'$.

Proof. Let us assume op was issued by process i and op' was issued by process k. Let $ts(op) = (i, T_1, \ldots, T_i, \ldots, T_n)$. From $op \rightarrow op'$, the response action of op happened before the invocation action of op'. So, the execution of the append operation $L_i.append(\langle ts, op\rangle)$ in the call $\mathsf{apply}(op, i)$ was completed before the $L_i.get()$ call in $\mathsf{apply}(op', k)$. Then, because of the linearizability of the ledgers, the length of ledger L_i found in $\mathsf{apply}(op', k)$ is $T'_i \geq T_i + 1$ (since the append operation increased its length). Hence, $op \prec op'$ from Definition 3. ◀

The proof of the next lemma is given in Appendix A.

▶ **Lemma 6.** *For each complete operation op (issued by i),* $\mathsf{valid}(\langle P(op), \prec\rangle, op, i) = True$. *Moreover, op returns in its response event the value* $\mathsf{execute}(\langle P(op), \prec\rangle, op, i)$.

▶ **Theorem 7.** *Code 4 implements a validated regular object as defined in Definition 1 in a crash-prone asynchronous system with linearizable SWMR DLOs.*

From the fact that reliable linearizable SWMR DLOs can be implemented in a crash-prone asynchronous system [18], we have the following corollary.

▶ **Corollary 8.** *It is possible to implement validated regular objects in an asynchronous system with n crash-prone processes from which up to $f < n/2$ can fail.*

4 Validated Totally-ordered Objects

4.1 Implementing Validated Totally-ordered Objects with Consensus

We consider now the set of validated totally-ordered objects as a whole. The first observation is that an object without validation can be seen as a validated object in which the $\mathsf{valid}()$ predicate always holds. Hence, an object with consensus number k [16] will also have at least consensus number k in its validated version.

Code 5 shows an algorithm that can be used to implement a validated totally-ordered object using an Atomic Broadcast service, which is known to be equivalent to Consensus [25] (and to MWMR Distributed Ledger Objects [1]). An Atomic Broadcast (AB) service [10, 6, 9, 22], ensures reliable and total ordering of the messages exchanged. Such a communication abstraction is based on appropriate crash-tolerant or Byzantine-tolerant consensus algorithms [10, 25].

The service has two operations, $AB.broadcast(m)$ used by a process to broadcast a message m to all other processes, and $AB.deliver(m)$ used by the service to deliver a message m to a process. From a user point of view, the AB service is defined by the following properties:

- *Validity*: if a correct process AB.broadcasts a message, it eventually AB.delivers it.
- *Agreement*: if a correct process AB.delivers a message, all correct processes will eventually AB.deliver that message.
- *Integrity*: a message is AB.delivered by a correct process at most once, and only if it was previously AB.broadcast.
- *Total Order*: the messages AB.delivered by the correct processes are totally ordered (i.e., if a correct process AB.delivers message m before message m', every correct process AB.delivers these message in the same order).

Note that if the AB service used is a crash-tolerant one, then Code 5 provides crash-tolerant implementation of the *apply* function, whereas if a Byzantine-tolerant AB service is used, then we have a Byzantine-tolerant implementation of *apply*. It follows that Code 5 implements a validated totally-ordered object defined by the $\mathsf{valid}()$ and $\mathsf{execute}()$ functions.

▶ **Theorem 9.** *Code 5 implements a validated totally-ordered object as defined in Definition 2 in a fault-prone asynchronous system with an Atomic Broadcast service.*

Proof. The claim holds from the following observations. Firstly, from the Agreement and Total Order properties of the AB service, all correct processes AB.deliver the same tuples $\langle op, j\rangle$ in the same order. This guarantees (by induction) that the sequence S maintained in all correct processes is the same. Moreover, for every $op \in S$ it holds that $\mathsf{valid}(S(op), op, j) = True$, where $S(op)$ is the subsequence preceding op in S. Finally, for every invocation $\mathsf{apply}(op, i)$ by a correct process i, the Validity of the AB service guarantees that the tuple $\langle op, i\rangle$ will be AB.delivered to i. Let $S_i(op)$ be the local value of the sequence S when op is AB.delivered to i. Then, the call $\mathsf{apply}(op, i)$ returns $(NACK, -)$ if $\mathsf{valid}(S_i(op), op, i) = False$, and it returns $(ACK, \mathsf{execute}(S_i(op), op, i))$ if $\mathsf{valid}(S_i(op), op, i) = True$. ◀

4.2 Persistent Validity

With Code 5 we have shown that all validated totally-ordered objects can be implemented with a Consensus / Atomic Broadcast service. In this section we explore conditions in the $\mathsf{valid}()$ and $\mathsf{execute}()$ functions that may allow a validated totally-ordered object to be implemented without consensus. We first define a property of some objects that we call *persistent validity.*

▶ **Definition 10.** *Given a validated totally-ordered object together with its valid() predicate, we say that the object satisfies* persistent validity *iff for every run R, with order $\lessdot$, every prefix S of $S(R)$*[2]*, and every operation $op_i \notin S$, if* $\mathsf{valid}(S, op_i, i) = True$ *then* $\nexists op_j \notin S, j \neq i : \mathsf{valid}(S, op_j, j) = True \wedge \mathsf{valid}(S||op_j, op_i, i) = False$.

Persistent validity informally says that once an operation is valid, then it cannot be made invalid by operations issued by the other processes. In Section 4.4 we show how *persistent validity* can help in the implementation of different objects according to different consistency criteria. But first, we investigate the implications of a validation predicate $\mathsf{valid}()$ for which the *persistent validity* does not hold.

[2] Recall that $S(R)$ is the sequence of operations in $C(R)$ totally ordered by $\lessdot$.

Code 6 Algorithm solving consensus for processes i and j when op_i and op_j invalidate each other.

```
Initialize object O with the prefix S
Init: cons_register_i and cons_register_j
    are atomic SWMR registers writable only
    by i and j respectively, initially ⊥.
Code for process i:
function PROPOSE(v_i)
    write(cons_register_i, v_i)
    r ← O.apply(op_i, i)
    if r = (NACK, −) then
        v_j ← read(cons_register_j)
        decide(v_j)
    else decide(v_i)
Code for process j:
function PROPOSE(v_j)
    write(cons_register_j, v_j)
    r ← O.apply(op_j, j)
    if r = (NACK, −) then
        v_i ← read(cons_register_i)
        decide(v_i)
    else decide(v_j)
```

4.3 Total Order Without Persistent Validity Is as Strong as Consensus

We demonstrate that a validated totally-ordered object whose valid() function does not satisfy the *persistent validity* property, is as strong as consensus. In order to do that, we will demonstrate that such an object can be used to solve the consensus problem between two crash-prone processes in an asynchronous system with at most one failure.

▶ **Observation 11.** *Let O be a validated totally-ordered object* without persistent validity. *Then, there is a run R of O, a prefix $S \subseteq S(R)$, and operations $op_i, op_j \notin S$ issued by processes $i \neq j$, such that* $\mathsf{valid}(S, op_i, i) = True \wedge \mathsf{valid}(S, op_j, j) = True \wedge \mathsf{valid}(S||op_j, op_i, i) = False$.

Informally, Observation 11 says that there is a run R' derived from R in which op_i issued by client i is valid if ordered after S, but there exists another valid operation op_j issued by a client $j \neq i$ that, if ordered before op_i, invalidates it. Note that no information is given on op_j, so it is not known if the inverse is true, i.e., whether op_i, if ordered before op_j, invalidates it.

We show now that object O, the prefix S, and the operations op_i and op_j can be used by processes i and j to reach consensus in an asynchronous system in which one of them can fail by crashing. Since without the object O it is known that in such a system consensus cannot be solved, we conclude that O is what allows to solve consensus.

In the rest of the section we hence assume a distributed system in which (at most) one process can crash, and computations happen in an asynchronous way so we can not make any assumption about processes relative speeds. The object O is assumed to be reliable, i.e. it does not fail or crash in any way. In addition, processes i and j, and the object O can use a reliable shared memory formed of atomic SWMR registers. As said before, such a shared memory can be implemented in an asynchronous message passing system if a majority of processes is correct [2, 18]. For our results to hold it is enough to assume that at most one process can crash, hence we assume $f = 1$. Then, while we focus on achieving consensus between processes i and j, if required, in order to implement the object O and the shared memory other processes can be involved. (In particular, at least a third process participates in the implementation of the shared memory to fulfill the requirement of a majority of correct processes.)

Both operations invalidate each other. Let us first assume that the two operations op_i and op_j are exclusive, i.e., if any one of the two is executed on the object O after prefix S, it makes invalid the other. In this case, Code 6 can be used by processes i and j to reach consensus. Observe that the code used by the two processes is completely symmetric.

▶ **Lemma 12.** *Let O be a validated totally-ordered object without persistent validity, and let prefix S, processes i and j, and operations $op_i, op_j \notin S$ as in Observation 11. Moreover,* *assume that* valid$(S||op_i, op_j, j) = False$*. Then Code 6 allows processes i and j to reach consensus.*

Proof. Process i first writes its proposed value v_i in its own register $cons\_register_i$ and then calls $O.apply(op_i, i)$. Process j does the same with register $cons\_register_j$ and call $O.apply(op_j, j)$. By assumption, only one of the operations op_i and op_j is found valid. Then, if process i receives an ACK from $apply(op_i, i)$, it can safely decide v_i, knowing that process j will receive $NACK$ and decide v_i as well. On the other hand, if process i receives $NACK$ and process j receives ACK, value v_j is decided by both processes.

Let us now assume that one process crashes; wlog, process j. If process j never issued the call $O.apply(op_j, j)$ or the call was issued but op_j was found invalid, then $O.apply(op_i, i)$ will return ACK and process i will decide v_i. If, on the other hand, j issued the call $O.apply(op_j, j)$ and op_j was found valid, then process i receives a $NACK$, and reads $cons\_register_j$. Since the value v_j was written in $cons\_register_j$ by j before calling $O.apply(op_j, j)$, the read operation completes and returns v_j, which is the value decided by process i. Process j cannot decide a different value, since $O.apply(op_j, j)$ returns ACK. ◀

Operation op_i does not invalidate operation op_j. We now deal with the case in which op_j makes op_i invalid, but op_i does not invalidate op_j. Observe that Code 6 does not solve this case because, since op_j is always valid, the value returned by call $O.apply(op_j, j)$ does not allow process j to know whether op_i was found valid. Notice that we use the validated totally-ordered object as a black box. Therefore, process j does not have direct access to the totally-ordered sequence of operations in the object. Thus, for process j to know whether op_i is found valid some extra work needs to be done. The key of the solution is the use of the shared memory available in the system to log the calls O.apply() and the values they return. To do so, a generic process k has a SWMR vector $oplist_k$ through with apply() call are issued. The result of the call is written by object O in a SWMR vector $reslist_k$ from where k can read it. This process is encapsulated in the side of the generic caller process k in the function LoggedApply() presented in Code 7.

Code 7 LoggedApply(op) function to communicate with O. It returns (s, r), where $s \in \{ACK, NACK\}$. Code for process k.

init: $oplist_k$ are SWMR vectors writable only by k, initially $\perp$
init: $reslist_k$ are SWMR vectors writable only by O, initially $\perp$
init: $c_k \leftarrow 1$ ▷ c_k is a local variable of k
function LoggedApply(op, k)
    write ($oplist_k[c_k], op, k$)
    wait until $reslist_k[c_k] \neq \perp$
    $res \leftarrow$ read ($reslist_k[c_k]$)
    $c_k \leftarrow c_k + 1$
    return (res)

On its side, object O is waiting for apply() calls being issued via the vector $oplist_k$, and when one appears it applies it and writes in $reslist_k$ the corresponding result. This can be implemented with one concurrent task for each process k as presented in Code 8. Note that, since the object O and the shared memory are both reliable, if an apply() call is written by process k in $oplist_k$, eventually the corresponding response will be written in $reslist_k$, even if k has crashed in the mid time.

Code 8 Task executed by object O to process the apply() calls issued by process k.

Init: $oplist_k$ and $reslist_k$ are the vectors from Code 7
init: $c_k \leftarrow 1$ ▷ c_k is a local variable of O
loop
    wait until $oplist_k[c_k] \neq \perp$
    $op \leftarrow$ read ($oplist_k[c_k]$)
    $res \leftarrow$ apply(op, k)
    write ($reslist_k[c_k], res$)
    $c_k \leftarrow c_k + 1$

With this logged method of using the object, the algorithm that processes i and j can use to solve consensus is presented in Code 9. Observe that the code for process i is similar to the one in Code 6, replacing the call O.apply(op_i, i) with call LoggedApply(op_i, i). However,

Code 9 Algorithm that solves consensus for processes i and j when op_i does not invalidate op_j.

```
1:  Initialize object O with prefix S
2:  Init: cons_register_i and cons_register_j
    are SWMR registers writable only by i and
    j respectively, initially ⊥
3:  Init: oplist_i and reslist_i are the vectors from
    Code 7
4:  Code for process i:
5:  function PROPOSE(v1)
6:      write (cons_register_i, v1)
7:      res ← LoggedApply(op_i, i)
8:      if res = (NACK, −) then
9:          v2 ← read (cons_register_j)
10:         decide(v2)
11:     else  decide(v1)
12: Code for process j:
13: function PROPOSE(v2)
14:     write (cons_register_j, v2)
15:     res ← LoggedApply(op_j, j)
16:     if ∃c : op_i = oplist_i[c] then
17:         wait until reslist_i[c] ≠ ⊥
18:         opires ← read (reslist_i[c])
19:         if opires = (ACK, r) then
20:             v1 ← read (cons_register_i)
21:             decide(v1)
22:         if opires = (NACK, −) then
23:             decide(v2)
24:     else   decide(v2)
```

the code for process j is different, since it has to access $oplist_i$ and $reslist_i$ to determine whether op_i was found valid.

▶ **Lemma 13.** *Let O be a validated totally-ordered object without persistent validity, and let prefix S, processes i and j, and operations $op_i, op_j \notin S$ as in Observation 11. Moreover, assume that* valid$(S||op_i,\ op_j, j) = True$. *Then Codes 8, 7, and 9 allow processes i and j to reach consensus.*

Proof. Without crashes, both processes i and j start by writing their proposed values v_i and v_j in their respective $cons\_register_i$ and $cons\_register_j$. Then, they call LoggedApply() with their operations op_i and op_j. As in Code 9, process i waits for response and decides v_i or v_j depending on whether op_i was found valid or not. This is determined from the value returned by the LoggedApply(op_i, i) call.

On its hand, process j always receives ACK from the LoggedApply(op_j, j) call, since operation op_j is found valid by hypothesis. So, it can not use this to know whether op_i precedes op_j and was hence found valid. Instead, it first checks if process i submitted op_i via a LoggedApply(op_i, i) call by searching in the $oplist_i$ vector. If op_i was not submitted, then process j can safely decide v_2 (line 24), because if it is submitted now it will be found invalid. Note that process i will decide v_2 as well.

If op_i is found in $oplist_i$ (line 16), then process j needs to wait for the result of LoggedApply(op_i, i) by reading from register $reslist_i$. As mentioned, because of the reliability of the object and the shared memory, the result will eventually be written there. At this point, if the result of LoggedApply(op_i, i) is ACK then it means that op_i was ordered before op_j, and the value to be decided is v_i. If it is $NACK$ then op_j has been ordered before op_i, op_i was invalid, and the value to be decided is v_j. In either case, the decided value is consistent with the one decided by process i, solving consensus between the two processes.

The correctness for the case when process j crashes is as in the proof of Lemma 12. If process i crashes before writing op_i in $oplist_i$, then j decide v_j as described above. Otherwise, op_i will be processed by O and found valid (and v_1 will be the decided value in both processes) or invalid (and v_2 will be the decided value). ◀

▶ **Definition 14.** *Given a validated totally-ordered object together with its associated* valid() *predicate and* execute() *function, we say that the object satisfies* persistent execution *iff*

1. *it satisfies persistent validity and*
2. *for every run R, with order $\lessdot$, every prefix S of $S(R)$, and every pair of operations $op_i, op_j \notin S$ from processes $i \neq j$, if* $\mathsf{valid}(S, op_i, i) = True \wedge \mathsf{valid}(S, op_j, j) = True$ *then* $\mathsf{execute}(S, op_i, i) = \mathsf{execute}(S||op_j, op_i, i)$.

▶ **Theorem 15.** *Let O be a validated totally-ordered object without persistent validity, then O can be used to solve consensus in a crash-prone asynchronous system with $n \geq 3$ processes in which at most one process can crash.*

Proof. From Lemmas 12 and 13 we have that two processes i and j can solve consensus between them. To make the solution applicable to the n processes, and allow any of the n values proposed to be decided, we have each process writing its proposed value in a SWMR register $prop_k$ in the shared memory. Processes i and j wait until $n-1$ such registers are filled, and choose one value from these values. Then, they run the consensus algorithm between them, proposing the chosen values. As soon as one of the two processes decides, it writes the decision in the shared memory. They use SWMR registers $decision_i$ and $decision_j$. Since at least one process i or j is correct, the value decided is eventually written in at least one of these registers. Then, the other processes can read it from there and also decide. ◀

Observe that in a crash-prone asynchronous system in which one process can crash consensus cannot be solved [11]. Thus, Theorem 15 implies that any validated totally-ordered object without persistent validity is as strong as a consensus object [24] in such a system.

4.4 Consensus-free Total Order with Persistent Execution

The previous result shows that persistent validity is required in order to be able to implement a validated totally-ordered object without consensus. Unfortunately this is not enough, as can be trivially observed from the fact that a $\mathsf{valid}()$ predicate that is always $True$ satisfies persistent validity. To be able to implement the object without consensus, some *additional* condition must be imposed. The following is an instance of such a condition.

▶ **Definition 16.** *Given a validated totally-ordered object together with its associated* $\mathsf{valid}()$ *predicate and* $\mathsf{execute}()$ *function, we say that the object satisfies* persistent execution *iff*

1. *it satisfies persistent validity and*
2. *for every run R, with order $\lessdot$, every prefix S of $S(R)$, and every pair of operations $op_i, op_j \notin S$ from processes $i \neq j$, if* $\mathsf{valid}(S, op_i, i) = True \wedge \mathsf{valid}(S, op_j, j) = True$ *then* $\mathsf{execute}(S, op_i, i) = \mathsf{execute}(S|op_j, op_i, i)$.

An object with persistent execution has significant flexibility for reordering concurrent operations to obtain different total orders $\lessdot$ that satisfy the conditions of Definition 2. Consider a validated totally-ordered object O and a finite run R. Let K be a set of concurrent operations issued by different processes k, such that $\forall op_k \in K$, it holds that $op_k \notin S(R)$, $\nexists op \in S(R) : op_k \to op$, and $\mathsf{valid}(S(R), op_k, k) = True$.

▶ **Lemma 17.** *If the validated totally-ordered object O satisfies persistent execution, then R can be extended with all the operations in K, in any order, satisfying Definition 2. Moreover, $\forall op_k \in K$, the value returned by $O.\mathsf{apply}(op_k, k)$ is $(ACK, \mathsf{execute}(S(R), op_k, k))$.*

Proof. Any extension R' as described will respect property (1) of Definition 2, because the operations in K do not precede in real time order those in $S(R)$ and they are concurrent among themselves. Regarding property (2), from the assumption that

$\forall op_k \in K, \mathsf{valid}(S(R), op_k, k) = True$, that the operations in K are issued by different processes, and persistent validity, we have that all operations in K will be valid in the extension of R. Finally, the value returned for op_k will be $(ACK, \mathsf{execute}(S(R), op_k, k))$ from property (2) of Definition 16. ◀

From this lemma, we can derive that Code 4 implements a validated totally-ordered object O when persistent execution is satisfied. The total order $\prec\!\!\prec$ of a run of O has to be an extension of the order from Def. 3, imposing an order among those operations that are not ordered by $\prec$. One possibility is to order complete operations in a run by the real time order of their response events in the history of the run. This total order is consistent with $\prec$ because Def. 3 and Code 4 guarantee that (1) if $op \prec op'$ then op completes before op' and that (2) if op completes before op' and $op \not\prec op'$ then op and op' are concurrent. Hence the following result, which implies that consensus is not required to implement validated totally-ordered objects with persistent execution.

▶ **Theorem 18.** *It is possible to implement a validated totally-ordered object O that satisfies persistent validity and persistent execution in an asynchronous system with n crash-prone processes from which up to $f < n/2$ can fail.*

5 Applications of Validated Objects

To demonstrate the usefulness of validated objects, in this section we present a number of possible applications providing the exact properties that each application satisfies. For each application, we present both a *relaxed* version, i.e, one that uses regular validated objects, and a *strict* version, i.e, one that uses totally-ordered validated objects, and we analyze what validity properties are required for the applications being realized. (One more application is given in Appendix B.)

5.1 Punching System

A punching system is an object that can be used by a process to log its activity. It essentially allows a process to signal the start of an activity and then signal that activity's end. One practical such system is used for tracking employee arrival and departure in various organisations. Such object may have the following two operations:

- $\mathsf{punch\text{-}in}(t, i)$, that can only be invoked by process i, to mark his arrival at time t,
- $\mathsf{punch\text{-}out}(i)$, that can only be invoked by process i, to mark his departure and return the hours worked since he last punched-in

Code 10 Functions valid() and execute() to implement a punching system object.

```
function valid(⟨P, ≺⟩, op, i)
    if (i is not the issuer of op) then
        return(False)
    lop_i ← {op' : op' the last operation of i in P}
    if (op = punch-out(k)) then
        return (i = k ∧ lop_i = {punch-in(t, i)})
    else                                         ▷ op = punch-in(t, k)
        return (i = k ∧ op = punch-in(t, k)∧
                (lop_i = ∅ ∨ lop_i = {punch-out(i)}))
function execute(⟨P, ≺⟩, op, i)
    if (op = punch-out(i)) then
        lt_i ← {t : op' = punch-in(t, i)∧
                ∄op'' ∈ P s.t. op' ≺ op''}
        return (hours(now() − lt_i))
    else return (⊥)
```

Notice that the $\mathsf{punch\text{-}in}(t, i)$ operation for i is only valid if the last operation from i was a $\mathsf{punch\text{-}out}(i)$ operation and vice-versa.

This object has both the persistent validity and persistent execution properties, as whenever i recorded a punch-in operation the punch-out operation remains valid no matter of the operations executed by any other process j. Persistent execution also holds since the value of the object at i remains the same until i performs its punch-in or punch-out operations.

Notice that since the process i is restricted to obtain its own working hours (i.e., invoke only punch-out(i)) then by well-formedness the relaxed version of the punching system is equivalent with the strict version. Thus, the system may be implemented without consensus utilizing the functions defined in Code 10. Recall in this code that P is the set of complete operations that precede op using the order $\prec$. Note also that $\prec$ orders all the operations from the same process (from well-formedness and property (1) of Definitions 1 and 2), so lop_i is well defined.

▶ **Theorem 19.** *Code 10 combined with Code 4 implements a strict punching system that satisfies both persistent validity and persistent execution.*

5.2 Cryptocurrency

In this section we implement a cryptocurrency (asset transfer) [15]. For that, a validated object is created, which holds an account for each process in $[1, n]$. For simplicity we assume that each process has initially a balance of *ibalance* tokens. The object has only two operations as described in [15]:

- transfer(i, k, x), that can only be invoked by process i, transfers $x > 0$ tokens from the account of the issuing process i to the account of process k, and
- read(k), which returns an estimate of the current balance of the account of process k.

We assume that the operations are cryptographically signed by the issuer. As usual, it is not allowed that a process ever has negative balance. Hence, an operation transfer(i, k, x) is valid and can be executed only when the balance of i is higher than the amount x to be transferred. In [15] this validation is embedded of the operation execution, while here validation and execution are separated in different functions valid() and execute() (see Code 11).

We can also get this object in two flavors. In the relaxed version of the object the value returned by the read(k) operation must include all operations that precede the read(k) in real time ordering, but may not include some of the transfer() operations that are concurrent with the call. Thus, the operation may return a lower bound of the actual balance (including the concurrent operations). On the other hand, in the strict version (i.e., if we use a validated totally-order object) then the balance operations will return the exact amount of the balance. The same applies to transfer() operations. In the relaxed version some of them may be found invalid because incoming funds in concurrent transfers are not accounted for.

Code 11 Functions valid() and execute() to implement a cryptocurrency.

function valid($\langle P, \prec\rangle, op, i$)
  if (i is not the issuer of op)$\vee$
    (signature of op is invalid) **then return** ($False$)
  if $op =$ read(k) **then return**($True$)
  else ▷ $op =$ transfer(j, k, x)
    if ($op =$ transfer(j, k, x) $\wedge j \neq i$) $\vee$ ($x \leq 0$) **then**
      return ($False$)
    $b_{in} \leftarrow ibalance + \sum\{x' : \exists j,$ transfer$(j, i, x') \in P\}$
    $b_{out} \leftarrow \sum\{x' : \exists j,$ transfer$(i, j, x') \in P\}$
    return ($b_{in} - b_{out} \geq x$)
function execute($\langle P, \prec\rangle, op, i$)
  if $op =$ read(k) **then**
    $b_{in} \leftarrow ibalance + \sum\{x' : \exists j,$ transfer$(j, k, x') \in P\}$
    $b_{out} \leftarrow \sum\{x' : \exists j,$ transfer$(k, j, x') \in P\}$
    return ($b_{in} - b_{out}$)
  else return ($\perp$) ▷ $op =$ transfer(i, k, x)

Code 12 Functions valid() and execute() to implement a Do-All object given a threshold T taken from a set J of jobs to execute.

```
function valid(⟨P, ≺⟩, op, i)
    if (i is not the issuer of op) then
        return(False)
    if (op = completed(x, k)) then
        return(i = k)
    else                                  ▷ op = do(x, k)
        if (op = do(x, k) ∧ i = k) then
            c ← |{j : do(x, j) ∈ P}|
            return(c ≤ T)
        else  return(False)

function execute(⟨P, ≺⟩, op, i)
    if (op = completed(x, k)) then
        return((do(x, k) ∈ P))
    else  return(⊥)
```

In order to implement the relaxed version of this object, it is enough to use the functions valid() and execute() as defined in Code 11, and use them in Code 4. Observe that the cryptocurrency object satisfies the property of persistent validity but it does not satisfy the property of a persistent execution. Therefore, in order to implement the strict version of this object, one may combine the functions of Code 11 with Code 5, which uses the Atomic Broadcast service.

▶ **Theorem 20.** *Code 11 combined with Code 4 or with Code 5, implement the relaxed and strict cryptocurrrency (asset transfer), respectively.*

5.3 Do-All: Task Execution

Do-All is an object in which a set of processes execute tasks taken from a set of jobs [13, 14]. Any process can take any task from the job set. In respect to the number of jobs that processes should execute, the specification of the Do-All can be traced to (1) a strict validated totally-ordered object, Definition 2, if a specific number T of job's executions must be respected, or (2) to a relaxed validated regular object, Definition 1, if executions of jobs beyond the threshold T can be tolerated when some conditions are met, e.g., if they were initiated in a batch of concurrent operations.

Specifically, the Do-All object supports the following operations:

- $Do(t, i)$: process i claims and performs task t.
- $Completed(t, i)$: process i reports the completion of task t.

$Do()$ is not valid if a certain number of processes performed the task (say 3 for redundancy). Notice that, as mentioned, we can have the strict version (i.e., totally-order version) of the object, in which *exactly* 3 processes can do a task, and the relaxed version (regular version) in which *at least 3* do it. Observe that this object does not satisfy the persistent validity property, nor the persistent execution one. Code 12 shows an implementation for the valid and execute predicates to realize the Do-All object in both cases. The following result holds.

▶ **Theorem 21.** *Code 12 combined with Code 4 or with Code 5, implement the relaxed and strict Do-All object, respectively.*

6 Conclusions

In this paper we have formalized the notion of a validated object, decoupling the object operations and properties from the validation procedure. We have focused on two type of objects, satisfying different levels of consistency: the validated totally-ordered object, offering a total ordering of its operations, and its weaker variant, the validated regular object. For both types, we have provided crash-tolerant implementations. Note that these implementations only attempt to prove that it is possible to implement different types of validated objects with and without consensus. Our objective was not to make them as efficient as possible; this is the subject of future work.

For validated totally-ordered objects, we further considered the persistent validity and persistent execution properties and their impact on the object's implementation. Our investigation has shown that (i) in the absence of persistent validity, the object is as strong as consensus, and (ii) persistent validity is not enough to implement a validated totally-ordered object without consensus; persistent execution was needed. An interesting future direction is to investigate whether there exists a weaker property than persistent execution, that together with persistent validity would yield consensus-free implementations of validated totally-ordered objects.

Furthermore, this investigation could be extended for Byzantine failures. We believe that with certain adjustments, a Byzantine-tolerant implementation of validated regular objects can be obtained from the one presented in Section 3. Observe that the consensus-based implementation of validated totally-ordered objects presented in Section 4.1 can tolerate Byzantine failures, when a Byzantine-tolerant Atomic Broadcast service is used. Also, the negative result of Section 4.3 trivially applies to Byzantine failures. What remains to be investigated are the conditions under which it is possible to obtain Byzantine-tolerant consensus-free implementations of validated totally-ordered objects. Finally, other consistency levels for validated objects can be defined, beyond regular and totally-ordered, and their implementability in different distributed system models be explored.

References

1 Antonio Fernández Anta, Kishori M. Konwar, Chryssis Georgiou, and Nicolas C. Nicolaou. Formalizing and implementing distributed ledger objects. *SIGACT News*, 49(2):58–76, 2018. URL: `http://doi.acm.org/10.1145/3232679.3232691`, `doi:10.1145/3232679.3232691`.

2 Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, 1995. `doi:10.1145/200836.200869`.

3 Hagit Attiya and Jennifer Welch. *Distributed computing: fundamentals, simulations, and advanced topics*, volume 19. John Wiley & Sons, 2004.

4 Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. *IACR Cryptol. ePrint Arch.*, page 6, 2001. URL: `http://eprint.iacr.org/2001/006`.

5 Vicent Cholvi, Antonio Fernández Anta, Chryssis Georgiou, Nicolas Nicolaou, and Michel Raynal. Atomic appends in asynchronous byzantine distributed ledgers. In *16th European Dependable Computing Conference, EDCC 2020, Munich, Germany, September 7-10, 2020*, pages 77–84. IEEE, 2020. `doi:10.1109/EDCC51268.2020.00022`.

6 Paulo Coelho, Tarcisio Ceolin Junior, Alysson Bessani, Fernando Dotti, and Fernando Pedone. Byzantine fault-tolerant atomic multicast. In *DSN 2018*, pages 39–50. IEEE, 2018.

7 Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. DBFT: efficient leaderless byzantine consensus and its application to blockchains. In *17th IEEE International Symposium on Network Computing and Applications, NCA 2018, Cambridge, MA, USA, November 1-3, 2018*, pages 1–8. IEEE, 2018. `doi:10.1109/NCA.2018.8548057`.

8 Tyler Crain, Christopher Natoli, and Vincent Gramoli. Red belly: A secure, fair and scalable open blockchain. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 466–483. IEEE, 2021. `doi:10.1109/SP40001.2021.00087`.

9 F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to byzantine agreement. *Information and Computation*, 118(1):158 – 179, 1995.

10 Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004. doi:10.1145/1041680.1041682.

11 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of ACM*, 32(2):374–382, 1985. doi:http://doi.acm.org/10.1145/3149.214121.

12 Davide Frey, Lucie Guillou, Michel Raynal, and François Taïani. Consensus-free ledgers when operations of distinct processes are commutative. In *International Conference on Parallel Computing Technologies*, pages 359–370. Springer, 2021.

13 Chryssis Georgiou and Alexander A. Shvartsman. *Do-All Computing in Distributed Systems: Cooperation in the Presence of Adversity*. Springer, 2008. doi:10.1007/978-0-387-69045-2.

14 Chryssis Georgiou and Alexander A. Shvartsman. *Cooperative Task-Oriented Computing: Algorithms and Complexity*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2011. doi:10.2200/S00376ED1V01Y201108DCT007.

15 Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. The consensus number of a cryptocurrency. In Peter Robinson and Faith Ellen, editors, *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 307–316. ACM, 2019. doi:10.1145/3293611.3331589.

16 Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991. doi:10.1145/114005.102808.

17 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

18 Damien Imbs, Sergio Rajsbaum, Michel Raynal, and Julien Stainer. Read/write shared memory abstraction on top of asynchronous byzantine message-passing systems. *J. Parallel Distributed Comput.*, 93-94:1–9, 2016. doi:10.1016/j.jpdc.2016.03.012.

19 Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97 — Object-Oriented Programming*, pages 220–242, 1997.

20 Leslie Lamport. On interprocess communication, part I: Basic formalism. *Distributed Computing*, 1(2):77–85, 1986.

21 N.A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

22 Zarko Milosevic, Martin Hutle, and André Schiper. On the reduction of atomic broadcast to consensus with byzantine faults. In *SRDS 2011*, pages 235–244, 2011.

23 Nicolas C. Nicolaou, Antonio Fernández Anta, and Chryssis Georgiou. Cover-ability: Consistent versioning in asynchronous, fail-prone, message-passing environments. In Alessandro Pellegrini, Aris Gkoulalas-Divanis, Pierangelo di Sanzo, and Dimiter R. Avresky, editors, *15th IEEE International Symposium on Network Computing and Applications, NCA 2016, Cambridge, Boston, MA, USA, October 31 - November 2, 2016*, pages 224–231. IEEE Computer Society, 2016. doi:10.1109/NCA.2016.7778622.

24 Michel Raynal. *Concurrent Programming - Algorithms, Principles, and Foundations*. Springer, 2013. doi:10.1007/978-3-642-32027-9.

25 Michel Raynal. *Fault-Tolerant Message-Passing Distributed Systems - An Algorithmic Approach*. Springer, 2018. doi:10.1007/978-3-319-94141-7.

26 Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. http://bitcoinaffiliatelist.com/wp-content/uploads/ethereum.pdf, 2014. Accessed: 2016-08-22. URL: http://bitcoinaffiliatelist.com/wp-content/uploads/ethereum.pdf.

A Proof of Lemma 6

Statement. For each complete operation op (issued by i), $\mathsf{valid}(\langle P(op), \prec\rangle, op, i) = True$. Moreover, op returns in its response event the value $\mathsf{execute}(\langle P(op), \prec\rangle, op, i)$.

Proof. The claim follows if we show that the set P created in Line 6 of Code 4 is the same as $P(op)$. Recall that $ts(op) = (i, T_1, \dots, T_k, \dots, T_n)$. Observe that in a ledger L_k the timestamp $ts = (k, T'_1, \dots, T'_k, \dots, T'_n)$ in the jth record in the ledger has $T'_k = j - 1$. Then, it holds that $P \subseteq P(op)$, because for each k, for each record $\langle ts', op'\rangle \in G_k$, the timestamp $ts' = (k, T'_1, \dots, T'_k, \dots, T'_n)$ satisfies that $T'_k < T_k = |G_k|$.

Let us assume there is an operation $op' \in P(op)$ (hence, $op' \prec op$) and $op' \notin P$. Assume op' was issued by process k, and $ts(op') = (k, T'_1, \dots, T'_k, \dots, T'_n)$. Then, by linearizability of the ledgers and $op' \notin P$, op' was appended in ledger L_k after the $L_k.get()$ of $\mathsf{apply}(op, i)$ found T_k records in the ledger. Hence, op' is the jth record in ledger L_k, where $j > T_k$. Note from Code 4 that $T'_k = j - 1$, since by well-formedness the jth operation of process k finds $j - 1$ records in ledger L_k. Then, $T'_k \geq T_k$, and hence it cannot happen that $op' \prec op$. ◀

B Versioned Read/Write Objects

A versioned object is a read/write object with the difference that each value written is associated with a version from a totally-ordered set of versions. A write operation succeeds only if it attempts to write a value with a version higher than any of the versions used by previous write operations; otherwise the write operation fails. In particular the object was introduced in [23], and supports two operations:

- $\mathsf{write}(\langle ver, v\rangle, x)$: process i attempts to write value v with version ver on object x.
- $\mathsf{read}(x)$: process i attempts to read the latest value and version of the object x.

In the strict case only the writes that satisfy the total ordering may be executed and thus this will ensure a strict order on the version of the writes. Therefore, we will obtain a single consistent sequence of versions. On the other hand on the relaxed case, multiple writes promoting the same version may conflict allowing multiple writes to be executed. In such a case only some of those will succeed by the operation definition, thus ensuring the properties of Coverability as presented in [23].

A versioned object does not satisfy persistent validity, neither persistent execution. Thus, in order to implement the strict version of the object we use consensus. The following result holds.

▶ **Theorem 22.** *Code 13 combined with Code 4 or with Code 5, implement the relax and strict versioned R/W object, respectively.*

Algorithm 13 Functions valid() and execute() to implement a R/W versioned object.

```
function valid(⟨P, ≺⟩, op, i)
    if (i is not the issuer of op) then
        return(False)
    if (op = read(x)) then
        return(True)
    else                                                    ▷ op = write(⟨ver, v⟩, x)
        if op = write(⟨ver, v⟩, x) then
            ver_max ← max {ver : write(⟨ver, ∗⟩, x) ∈ P}
            return(ver > ver_max)
        else
            return(False)

function execute(⟨P, ≺⟩, op, i)
    if (op = read(x)) then
        ver_max ← max {ver : write(⟨ver, ∗⟩, x, j) ∈ P}
        v_max ← {v : write(⟨ver_max, v⟩, x, j) ∈ P}
        return(⟨ver_max, v_max⟩)
    else
        return(⊥)
```