Formalizing Distributed Ledger Objects

Antonio Fernández Anta [*] Chryssis Georgiou [†] Nicolas Nicolaou [‡]

In his PODC'2017 keynote address, Maurice Herlihy pointed out that despite the hype about blockchains and distributed ledgers, no formal abstraction of these objects has been proposed. To face this issue, in this paper we provide a proper formulation of a *distributed ledger object*. In brief, we define a *ledger* object as a sequence of *records*, and we provide the operations and the properties that such an object should support. We then provide a variation of the ledger – the *validated ledger* – which requires that each record in the ledger satisfies a particular *validation rule*. A (validated) ledger is *distributed* if it is implemented on top of multiple (possibly geographically dispersed) *computing devices*.

Concurrent Objects. 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 [3]. Each operation on an object O consists of *invocation* and *response* events. A *history* of operations on O, denoted by H_O, is a sequence of invocation and response events, starting with an invocation event. 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. 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 history H_O is *sequential* if all operations in H_O are complete and it contains no concurrent operations (i.e., it is an alternative sequence of matching invocation and response events, starting with an invocation and ending with a response event). A *sequential specification* of an object O, defines the behavior of O in every sequential history H_O (i.e., when accessed sequentially) [3].

(Validated) Ledger Object. A *ledger* $\mathcal{L}$ is a concurrent object that stores a totally ordered sequence[1] of *records* and supports two operations (available to any process p): (i) $\mathsf{get}()_p$, and (ii) $\mathsf{append}(r)_p$. A *record* is a triple $r = \langle \tau, p, v \rangle$, where τ is a *unique* record identifier from a set $\mathcal{T}$, $p \in \mathcal{P}$ is the identifier of the process that created record r, and v is the data of the record drawn from an alphabet V. A process p invokes a $\mathsf{get}()_p$ operation[2] to obtain the ledger $\mathcal{L}$, and p invokes an $\mathsf{append}(r)_p$ operation to extend $\mathcal{L}$ with a new record r. Initially, the ledger $\mathcal{L}$ is empty.

Code 1 External Interface of a Ledger Object Executed by a Process p

```
1: function GET
2:     send request (GET) to the ledger
3:     wait response (GETRES, $\mathcal{L}$) from the ledger
4:     return $\mathcal{L}$

5: function APPEND($r$)
6:     send request (APPEND, $r$) to the ledger
7:     wait response (APPENDRES, $res$) from the ledger
8:     return $res$
```

A process p interacts with a ledger by invoking an operation (a $\mathsf{get}()_p$ or an $\mathsf{append}(r)_p$), which causes a request to be sent to the ledger, and waiting for a response, which marks the end of the operation[3]. The response carries the result of the operation. The result for a get operation is a sequence of records, while the result for an append operation is a confirmation (ACK). This interaction from the point of view of the process p is depicted in Code 1. A possible centralized implementation of the ledger that processes requests sequentially is presented in Code 2 (each block **Upon** is assumed to be executed in mutual exclusion).

[*] IMDEA Networks Institute, Madrid, Spain, antonio.fernandez@imdea.org

[†] Dept. of Computer Science, University of Cyprus, Nicosia, Cyprus, chryssis@cs.ucy.ac.cy

[‡] KIOS Research CoE, University of Cyprus, Cyprus, nicolasn@ucy.ac.cy

[1] For simplicity we will treat the ledger and the sequence it stores interchangeably, unless otherwise stated.

[2] We define only one operation to access the value of the ledger for simplicity. In practice, other operations, like those to access individual records in the sequence, will also be available.

[3] We make explicit the exchange of request and responses between the process and the ledger to reveal the fact that the ledger is concurrent, i.e., accessed by several processes.

Code 2 Ledger (centralized and sequential)

```
Init: $\mathcal{L} \leftarrow \emptyset$

Upon receiving request (GET) from process $p$ do
    send response (GETRES, $\mathcal{L}$) to $p$

Upon receiving request (APPEND, $r$) from process $p$ do
    $\mathcal{L} \leftarrow \mathcal{L} \| r$
    send response (APPENDRES, ACK) to $p$
```

Code 3 Validated Ledger ($\mathit{Valid}()$)

```
Init: $\mathcal{L} \leftarrow \emptyset$

Upon receiving request (GET) from process $p$ do
    send response (GETRES, $\mathcal{L}$) to $p$

Upon receiving request (APPEND, $r$) from process $p$ do
    if $\mathit{Valid}(\mathcal{L} \| r)$ then
        $\mathcal{L} \leftarrow \mathcal{L} \| r$
        send response (APPENDRES, ACK) to $p$
    else
        send response (APPENDRES, NACK) to $p$
```

A *validated ledger* is a ledger in which specific semantics are imposed over the sequence of the records stored in the ledger. For instance, if the records are (bitcoin-like) financial transactions, the semantics should, for example, prevent double spending. The ledger preserves the semantics with a validity check in the form of a Boolean function $\mathit{Valid}()$ that takes as an input a sequence of records S and returns true if and only if the semantics are preserved. In a validated ledger the result of an $\mathsf{append}(r)_p$ operation may be NACK if the validity check fails, as depicted in Code 3.

For instance, one possible semantic we may want to impose is that the same record identifier does not appear twice in the sequence. This can be achieved with the validated ledger and the $\mathit{Valid}()$ function in Code 4. More complex validation rules can be defined in a similar manner, e.g., the transaction validation used as part of the Bitcoin protocol [2].

Code 4 Duplication of Record IDs Validation

```
function $\mathit{Valid}(S)$
    if $\exists r, r' \in S$, such that $r.\tau = r'.\tau$ then
        return false
    else
        return true
```

Observe that the ledger implementation presented in Code 2 processes operations sequentially. Hence, it is possible to define a sequential history H of any execution in which operations appear in the order they are processed by the ledger. The *sequential specification* of a ledger over the sequential history H is defined as follows. If $\mathcal{L}$ is the value of the ledger after the invocation event in H of an operation π then: (a) if π is a get_p operation, then the response event of π returns $\mathcal{L}$, (b) if π is an $\mathsf{append}(r)_p$ operation that returns ACK, then at the response event of π the value of the ledger is $\mathcal{L} \| r$, and (c) if π is an $\mathsf{append}(r)_p$ operation that returns NACK, then at the response event of π the value of the ledger is $\mathcal{L}$.

Distributed (Validated) Ledger. A *distributed (validated) ledger* is a concurrent (validated) ledger object that is implemented in a distributed manner. In particular, the ledger object is *implemented* by (and possibly replicated among) a set of (possibly distinct and geographically dispersed) computing devices, that we refer as *hosts*. Distribution and replication intend to ensure availability and survivability of the ledger, in case a subset of hosts fail. At the same time, they raise the challenge of maintaining *consistency* among the different views that different processes get of the distributed ledger: what is the latest value of the ledger when multiple processes may send operation requests at different hosts concurrently? Consistency semantics needs to be in place to precisely describe the allowed values that a get operation may return when it is executed concurrently with other get or append operations. Here, as an example, we provide the properties that operations must satisfy in order to guarantee *atomic* consistency semantics. In a similar way, weaker consistency guarantees, such as sequential, causal, or eventual consistency can be defined (which we omit due to lack of space).

Atomicity (linearizability) [1] provides the illusion that the distributed ledger is accessed sequentially, even when operations are invoked concurrently. I.e., the distributed ledger seems to be a centralized sequential ledger like the one implemented by Code 2. Formally, a distributed ledger $\mathcal{L}_a$ is *atomic* if, given a set of complete operations Π, any two operations $\pi_1, \pi_2 \in \Pi$ satisfy the following properties: (a) if $\pi_1 = \mathsf{append}(r_1)$, $\pi_2 = \mathsf{append}(r_2)$, and $\pi_1 \to \pi_2$, then no get operation returns a ledger $\mathcal{L}_a$ in which record r_2 appears before r_1; (b) if $\pi_1 = \mathsf{append}(r)$, $\pi_2 = \mathsf{get}()$, and $\pi_1 \to \pi_2$, then r is in the ledger $\mathcal{L}_a$ returned by π_2; (c) if π_1 and π_2 are get operations that return $\mathcal{L}_1$ and $\mathcal{L}_2$ respectively, then $\mathcal{L}_1$ is a prefix of $\mathcal{L}_2$ or vice versa; moreover, if $\pi_1 \to \pi_2$ then $\mathcal{L}_1$ is a prefix of $\mathcal{L}_2$.[4]

Within our framework, we can devise implementations of atomic distributed ledgers by using atomic broadcast or consensus primitives (not presented due to lack of space).

[4]Observe that our definitions are independent of a specific communication medium; i.e., they apply, for example, in both message-passing and shared memory settings.

Acknowledgements. We would like to thank Kishori Konwar, Paul Rimba and Neha Narula for helpful discussions.

References

[1] 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.

[2] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[3] Michel Raynal. *Concurrent Programming: Algorithms, Principles, and Foundations*. Springer, 2013.