Distributed Ledger Register: From Safe to Atomic

Emmanuelle Anceaume[1], Marina Papatriantafilou[2], Maria Potop-Butucaru[3], and Philippas Tsigas[2]

[1] CNRS, Univ Rennes, Inria, IRISA, France
[2] Dept. of Computer Science and Engineering (CSE), Chalmers University of Technology, Sweden
[3] LIP6, Sorbonne University, France

Abstract. This paper continues the recent line of academic effort dedicated to formalizing distributed ledgers. This work is the first one to propose a specification of distributed ledger register that matches the Lamport hierarchy from safe to atomic. Moreover, we propose implementations of distributed ledger registers with safe, regular and atomic guarantees in a model of communication specific to distributed ledgers technology. Furthermore, we close an open problem pointed out in [2] related to the specification and the implementation of an atomic register designed to fully capture the specificities of permisionless blockchain systems.

1 Introduction

The blockchain is probably one of the hottest and most transformative topic in the current digital landscape. The bitcoin crypto-currency [19], the leading application of the blockchain technology, has shown to be remarkably stable, and as such is often cited as the universal solution for managing a broad range of information.

A permissionless blockchain (or distributed ledger) is commonly presented as "an immutable distributed ledger with decentralised control", i.e., a continuously growing list of records that mimics the functioning of a traditional ledger, namely transparency and falsification-proof of documentation. However, a distributed ledger evolves in an untrusted and open environment. By untrusted, one means an environment in which participants cannot rely on a (shared or centralized) third authority to check and validate all the information that is contained in the blockchain. By open, one means an environment in which (non trustworthy) participants are free to join and leave the system at any time, and as often as they wish. To guarantee transparency and falsification-proof of information, the blockchain must be publicly accessible and consistently updated by every participant.

Distributed ledgers, beyond their incontestable qualities such as decentralisation, simple design and relatively easy use, are neither riskless nor free of limitation. An increasing number of areas promote the use of distributed ledgers

for the development of their applications, and undeniably, the properties enjoyed by these technologies should be studied to fit such applications requirements, together with their relationships with blockchain-based applications.

Bridges between the distributed computing theory and distributed ledgers have been pioneered by Garay et *al* [10]. The main focus of the distributed community [6, 7, 9, 10, 13, 16, 22] has so far been the distributed ledger agreement aspects. Our paper investigates consistency properties of the distributed ledger technology to identify connections between this technology and read-write registers. This connection is important because it will help us argue about the correctness of the ledger itself but also for the applications that will use it. We thus do need to deeply understand and formulate the properties of the state of distributed ledgers. In a previous work [2], it was proven that the state of popular blockchain ledgers can be described as regular read-write registers. Regular registers provide weak guarantees that are not easy for the application programmer to use for building correct applications on top of those registers. More significantly they do not compose. Atomicity or linearizability is a state property that is composable [12], and thus would allow the combination of multiple distributed ledgers, and has clear strong semantics. Atomicity has been adopted as the standard property to have in the development of parallel and distributed systems. Previous work [2] left open the question of proposing distributed ledger registers with atomic semantics. In [2] the authors prove that Bitcoin and Ethereum verify only the specification of a distributed ledger register with regular semantics.

Our contributions. This paper extends the work of [2] in several ways. First, we propose a specification of a distributed ledger register that matches the Lamport hierarchy [17] from safe to atomic in a Byzantine prone environment. Then we propose implementations of the distributed ledger register that verify safe, regular and atomic semantics assuming the presence of Byzantine nodes. The model of communication is specific to distributed ledgers technology [8]. Specifically, the underlying system provides a broadcast primitive satisfying the Δ-delivery property (if a node invokes $\mathsf{broadcast}(m)$ then every correct node eventually $\mathsf{delivers}$ m within Δ time units). Finally, we propose an implementation of a distributed ledger register that satisfies the atomic specification and the k-consistency property defined in [2] to mimic permissionless distributed blockchains.

Paper Roadmap. Section 2 presents briefly the characteristics of permissionless blockchains (e.g Bitcoin or Ethereum). Section 3 presents the state of the art closely related to our work. Section 4 presents the computational model. Section 7 provides a brief summary of shared registers and their specification. In Section 5.2 we specify the Distributed Ledger Register and propose implementations of Distributed Ledger Register satisfying safe, regular and atomic semantic. Section 6 makes the connection between the Distributed Ledger Register specification defined in Section 5.2 and distributed ledgers and closes the open question proposed in [2]. Section 7 concludes and presents open problems.

2 Permissionless Blockchains

A permissionless blockchain is commonly presented as "an immutable append only ledger publicly readable and writable". By immutable append only ledger it is meant a continuously growing list of appended records (or blocks) that mimics the functioning of a traditional ledger, namely transparency and falsification-proof of documentation. Publicly readable and writable means that at any time any entity is allowed to (cryptographically) append new blocks in the ledger and in reading the entire sequence of records appended so far without being authorized by any trustworthy third entity.

From an implementation point of view, each entity of the system locally maintains its own copy of the ledger. Each newly created block cryptographically depends on the current local ledger. Once created, it is propagated within the system. Since blocks can be created concurrently, two or more blocks can reference the very same parent block, and hence the creation of a tree with several branches (chains). This situation is known as *ledger fork*. In a chain of blocks, each block references an earlier block by inserting a cryptographic link to this earlier block in its header. This forms a tree of blocks, rooted at the genesis block, in which a branch is a path from a leaf block to the root of the tree. Each branch, taken in isolation, represents a consistent history of the crypto-currency system, that is, does not internally contain any conflicting transactions, i.e., double-spending transactions. On the other hand, any two branches of the tree do not need to be consistent with each other. The reason is that, at any time, each node of the system selects the *best chain*, an unique branch to represent the history of the crypto-system. In Bitcoin [19], the best chain corresponds to the branch (starting from the genesis block) that required the most important quantity of work. When a transaction T is included in a block b, it is said *confirmed* by all the nodes that accept that block in their local copy of the blockchain. The *level of confirmation* of transaction T is the number of blocks included in the blockchain after b, b included; by extension, a 0 confirmation level means that the transaction has not yet been included in the blockchain. Nakamoto [19] as well as subsequent studies [10, 15, 18] have shown that if the proportion of malicious miners μ is equal to 10%, then with probability less than 0.1%, a transaction can be rejected if its level of confirmation in a local copy of the blockchain is less than 5. In the following, we say that a transaction is *deeply confirmed* once it reaches such confirmation level.

3 Related works

The first effort in specifying the properties of permissionless blockchain systems is due to Garay and Kiayias [10]. They characterized Bitcoin blockchain via its quality and its common prefix properties, i.e., they define an invariant that this protocol has to satisfy in order to verify with high probability an eventually consistent prefix. This line of work has been continued by [21].

In order to model the behavior of distributed ledgers at runtime, Girault et al. [11] present an implementation of the Monotonic Prefix Consistency (MPC)

criterion and showed that no criterion stronger than MPC can be implemented in a partition-prone message-passing system. On the other hand, the proposed formalization does not propose weaker consistency semantics more suitable for proof-of-work blockchains as BitCoin. In the same line of research, in [3], Anceaume et al. propose a new data type to formally model distributed ledgers and their behavior at runtime. They provide consistency criteria to capture the correct behavior of current blockchain proposals in a unified framework. Briefly, a distributed ledger is the composition of two finite state automata enriched with a consistency criteria that specifies the behavior of the ledger in presence of concurrency. The first automaton, called *blocktree abstract data type*, describes the transition of the tree of blocks of when read and append of new blocks are executed. The second automaton, called *token oracle*, captures the cryptographical process, *proof-of-work*, specific to permissionless ledgers that condition the append of new blocks. Furthermore, [3] proposes necessary conditions to implement a *blocktree abstract data type* and the study of consensus number of *token oracle*. This line of research paves the way to automatic design of distributed ledgers. In parallel and independently of [3], Anta et al [5] propose a formalization of distributed ledgers modeled as an ordered list of records. The authors propose three consistency criteria: eventual consistency, sequential consistency and linearizability. Interestingly, they show that a distributed ledger that provides eventual consistency can be used to solve the consensus problem. These findings confirm the results of [3] about the necessity of Consensus to solve the strong prefix consitency.

Our work is in the line of effort opened by [2] to connect distributed ledgers and distributed shared objects (registers) theory. The authors introduce the notion of Distributed Ledger Register (DLR) where the value of the register has a tree topology instead of a single value as in the classical theory of distributed registers. The vertices of the tree are blocks of transactions cryptographically linked. Further, the authors of [5] defined a ledger object as a totally ordered sequence of blocks (or records). This was extended in [4] by defining Multi-Distributed Ledger Objects (MDLO), which is the result of aggregating multiple Distributed Ledger Objects satisfying the definition proposed in [5]. This formalisation is appropriate for permissioned blockchains. The formalisation proposed in the current work is more general, since it covers both permissioned and permissionless ledgers. Compared with [2], where the DLR register properties were crafted to fit the permissionless blockchains behavior such as Bitcoin and Ethereum, the curent work refines the specifications in order to cover weaker semantics (i.e. safe shared objects) and stronger semantics (i.e. atomic shared objects). We also propose an implementation of Distributed Ledger with atomic semantics, problem left open in [2].

4 Computational Model

We model the system as a synchronous distributed system composed of an arbitrary finite but unknown number of processes.

Each process is a state machine, whose state, called "local state", is defined by the current values of its local variables. A configuration of the system is composed of the local state of each process in the system, and the set of messages that have been sent but not yet delivered.

It is assumed that the system has a built-in communication abstraction that allows processes to communicate by exchanging messages via a broadcast primitive. This primitive provides to operations, namely the broadcast() and the deliver() operations. This communication abstraction is defined by the following properties.

- Δ-*delivery*. There exists $\Delta > 0$ such that if a process invokes broadcast(m) then all correct processes deliver m within Δ time units.
- *Validity*. If a correct process delivers a message m from p then p has previously invoked broadcast(m).

We assume that the system does not partition. By this we mean that processes which are online will receive any broadcast message within Δ time units. For processes which are temporarily off-line (for more than Δ time units) they will receive any broadcast messages once they will be on-line again.

This paper focuses on the case where processes may commit *Byzantine failures*, i.e., may behave in a way that does not respect their intended behavior (as defined by their specification): it can crash, fail to send or receive messages, send arbitrary messages, start in an arbitrary state, perform arbitrary state transitions, etc. A process that commits no failure (i.e., a non-Byzantine process) is also called a correct process. Note that during the time a process is off-line, it is considered as incorrect. It should be noted that we do not limit the number of incorrect processes in the system.

5 From Classical Distributed Registers to Distributed Ledger Registers

5.1 Background on Distributed Registers

This section recalls the main properties of classical distributed read-write registers.

A distributed read-write register REG is a shared variable accessed by a set of processes through two operations, namely REG.write() and REG.read(). Informally, the REG.write() operation updates the value stored in the shared variable while the REG.read() obtains the value contained in the shared variable. Every operation issued on a register is, generally, not instantaneous and can be characterised by two events occurring at its boundaries: an *invocation* event and a *reply* event. Both events occur at two different instants with respect to the fictional global time: the invocation event of an operation op (i.e., $op = REG$.write() or $op = REG$.read()) occurs at the invocation time denoted by $t_B(op)$ and the reply event of op occurs at the reply time denoted by $t_E(op)$.

Given two operations op and op' on a register, we say that op *precedes* op' ($op \prec op'$) if and only if $t_E(op) < t_B(op')$. If op does not precede op' and op' does not precede op, then op and op' are *concurrent* (noted $op||op'$).

An operation *op* is *terminated* if both the invocation event and the reply event occurred (*i.e.*, the process executing the operation does not crash between the invocation time and the reply time). We suppose that a terminated operation can either be successful and thus returns *true* or can return *abort* when, for example, some operational conditions are not met [1]. More details appear in [2]. On the other hand, an operation that does not terminate is called *failed*.

When read() and write() operations concurrently access a shared variable, the meaning of *last written value* becomes ambiguous. Depending on the semantics of the operations, three types of registers have been defined by Lamport [17], i.e., *safe*, *regular* and *atomic*. Due to space limitations we recall in the Appendix the specifications of these registers when processes are subject to Byzantine faults.

5.2 Distributed Ledger Register

Interestingly enough, neither safety, regularity nor atomicity of distributed shared registers can be used to implement the behaviour of distributed ledgers. Classically, values written in a register are potentially independent, and during the execution, the size of the register remains the same. In contrast, a new block cannot be written in the blockchain if it does not depend on the previous one, and successive writings in the blockchain increase its size. Actually, blockchains behavior has some similarities with the stabilizing registers, line of research pioneered by [14]. Looking at the stabilizing register, it implements some type of eventual consistency, in the sense that, there exists a bounded prefix of the system execution for which there are no guarantees on the value of the shared register. In contrast, in blockchain systems, the prefix of the blockchain eventually converges at every process, while no guarantees hold for the last created blocks. In [20], a register tailored for a totally ordered sequence of versions has been proposed. Inspired by this work, in [2] the authors proposed a new register called the *Distributed Ledger Register* (DLR) crafted for permissionless blockchain systems such as Bitcoin or Ethereum. In the following we refine the expressiveness of the *Distributed Ledger Register* (DLR) in order to match as closely as possible the classical distributed registers hierarchy.

The Distributed Ledger Register (DLR) has a tree structure, whose root is the genesis block, and where each branch (in the following we will prefer the term "chain") is a totally ordered sequence of blocks. The *value* of DLR is the best chain of the tree. It is called a *blockchain* and is denoted by $\mathcal{B}$. The best chain of the tree is obtained by applying a function, best_chain(), to the tree. best_chain() is known by all processes. Specifically,

- The best_chain() function returns the sequence of blocks, starting from the root, which has required the most important quantity of work.

Each node has also access to the update_tree() function defined as follows.

- The update_tree() has two parameters: a blockchain $\mathcal{B}$ and a set of blocks W. For each block b in W, if b correctly matches the last block b' of blockchain $\mathcal{B}$ (that is, b is well-formed and cryptographically links b'), then b is appended

to b', and removed from W. If all the blocks in W have been appended then update_tree($\mathcal{B}$,W) returns true, $\perp$ otherwise.

We now present the specifications of safe, regular and atomic DLRs, and for each of them, we present an message-passing implementation assuming the computational model defined in Section 4. As with traditional registers, a DLR is equipped with write and read operations. The DLR.write operation invoked with block b updates the value of DLR with block b, while the DLR.read() operation returns the value of DLR, that is the blockchain of DLR.

5.3 Safe SWMR Distributed Ledger Register

The DLR.read() and DLR.write() operations of a *safe* SWMR distributed ledger register DLR satisfy the following two properties.

- *Termination*: Any invocation of DLR.write() or DLR.read() eventually terminates.
- *Validity*: A DLR.read() operation invoked by a correct node returns a blockchain $\mathcal{B}$ such that $\mathcal{B}$ contains the block of the last valid DLR.write operation that happened before DLR.read() invocation, or any value of the register domain in case the DLR.read() operation is concurrent to a DLR.write() operation. A DLR.write operation is valid if *(i)* its execution follows the DLR.write specification, and *(ii)* does not abort.

We now propose a message-passing implementation of the SWMR DLR. Pseudocode of the implementation is presented in Figure 1. Each process p locally maintains its own copy DLR_p of the Distributed Ledger Register, and accesses it through DLR_q.read() operation. An unique process, q (*a.k.n* writer), among the n ones can access its DLR_q with both DLR_q.read() and DLR_q.write() operations.

The implementation presented in Figure 1 is correct if both the termination and safety properties hold.

Lemma 1. *Let $\mathcal{B}$ be the blockchain returned at Line 02 of the* $\mathsf{DLR_q}$.write() *operation in Figure 1 at time $t \geq 0$ when q invokes the* best_chain *function. If q is correct, then by time $t' = t$ all the other correct processes have executed Line 10 of the algorithm, and the returned blockchain $\mathcal{B}'$ is such that $\mathcal{B}' = \mathcal{B}$.*

Proof. The proof is by induction on time t.

- *Base case:* at time $t = 0$, the local register DLR_p of each correct process p is initialized with the genesis block.
- *Inductive case:* By inductive hypothesis, we have that if at time t', $\mathcal{B}$ is the blockchain returned at Line 02 when q invoked the best_chain function, then by time t', all the other correct processes have executed Line 10 and have obtained $\mathcal{B}$ as blockchain. Since by assumption q is correct, if at time t', q invoked the best_chain function, then at time t', q was executing $\mathsf{DLR_q}$.write(b) operation. Now let t be the first time after time t' at which q invokes the best_chain function. Let $\mathcal{B}_1$ be the blockchain returned by best_chain. Two cases have to be considered:

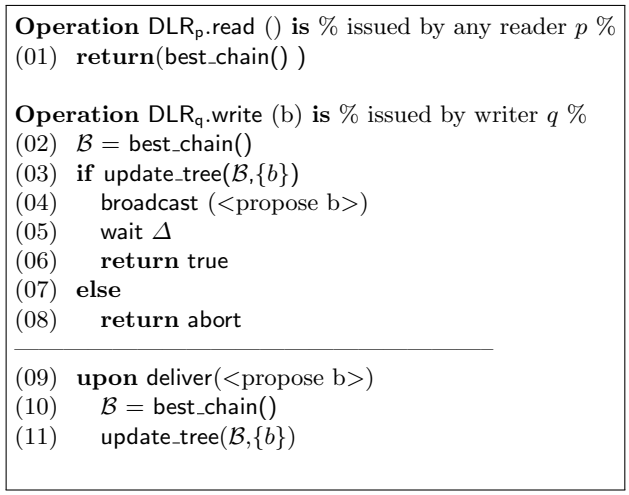

Fig. 1. DLR.read() and DLR.write() operations of the SWMR Safe Distributed Ledger Register.

- $\mathsf{DLR_q.write}(b)$ operation invoked at time t' is successful (i.e. returns true): At time t' the call to $\mathsf{update\_tree}(\mathcal{B}, \{\mathsf{b}\})$ function is successful, i.e., b is appended to $\mathcal{B}$, and then by time $t' + \Delta$ all correct processes have received b. Since by assumption of the inductive case they have the same blockchain $\mathcal{B}$ as q, then at time $t'+\Delta$ the $\mathsf{update\_tree}(\mathcal{B}, \{\mathsf{b}\})$ is successful, i.e., b is appended to $\mathcal{B}$. Now since q is correct, $t > t' + \Delta$. Thus at time t when q invokes the best_chain function, the returned blockchain is $\mathcal{B}$ to which has been appended b, which concludes this case.
- $\mathsf{DLR_q.write}(b)$ operation invoked at time t' returns abort: the case holds by the inductive case since block b has not been appended.

Lemma 2. *Suppose writer q is correct. If q invokes $\mathsf{DLR_p.write}(b)$ operation w at time t and w is valid, then by time $t + \Delta$, for any two correct processes p_i and p_j, DLR_i's value is equal to DLR_j's value.*

Proof. Straightforward from Lemma 1

Lemma 3. *The algorithm presented in Figure 1 implements a SWMR Safe Distributed Ledger Register.*

Proof. The Termination property is straightforward. For any correct process p, none of the $\mathsf{DLR_p.read}()$ and $\mathsf{DLR_p.write}()$ operations are blocking operations. Indeed, $\mathsf{DLR_p.read}()$ returns immediately after the best_chain invocation, while $\mathsf{DLR_q.write}()$ operation returns Δ time units after the broadcast invocation (with a returning code *true*) or immediately if the written block is not valid. In this case the returned code is *abort*.

For the Validity property, we distinguish two cases: the writer, q, is correct and the writer, q, is Byzantin.

- q is correct. Let r be a read operation $\mathsf{DLR_p.read}()$ invoked at time $t1$ by some process p, and w be the last valid $\mathsf{DLR_q.write}(b)$ operation executed by q before r. Let $t < t_1$ be the time at which w completes. Then by Lemma 2, all correct processes have the same blockchain, which contains b has last block. This completes this case of the proof.
- q is Byzantine. Let $w_1, \ldots, w_\ell$ be the largest sequence of valid writes operations performed by q. If such a sequence does not exist, then any $\mathsf{DLR_p.read}()$ operation will return a blockchain that starts with the genesis block, and thus safety holds. Otherwise, by Lemma 2, once the valid $\mathsf{DLR_q.write}(b)$ operation w_ℓ completes, for all correct processes p, the value of DLR_p contains the block written by w_ℓ, which concludes the proof.

5.4 Regular SWMR Distributed Ledger Register

The DLR.read() and DLR.write() operations of a *regular* SWMR distributed ledger register DLR satisfy the following two properties.

- *Termination*: Any invocation of DLR.write() or DLR.read() eventually terminates.
- *Validity*: A DLR.read() operation invoked by a correct node returns a blockchain $\mathcal{B}$ such that $\mathcal{B}$ contains the block of the last valid DLR.write operation that happened before DLR.read() invocation, or the last block written by a valid DLR.write() operation concurrent with the DLR.read() operation. A DLR.write operation is valid if *(i)* its execution follows the DLR.write specification, and *(ii)* does not abort.

We propose a message-passing implementation of a SWMR Distributed Ledger Register that satisfies the regular semantics. Pseudocode of the implementation is presented in Figure 2. Each process p locally maintains its own copy DLR_p of the Distributed Ledger Register, and accesses it through operation, while a single process q among the n ones can access its DLR_q with both the DLR_q.read() and the DLR_q.write() operations.

The difference between the implementation presented in Figure 2 and the one presented in Figure 1 is that the DLR.read() operation waits for Δ time units before returning the value of the DLR register, while the DLR.write() operation does not wait anymore after having broadcast the block to be appended to the DLR's value.

Lemma 4. *The implementation presented in Figure 2 implements a SWMR Regular Distributed Ledger Register.*

Proof. Termination of both operations holds by just switching the arguments in the liveness part of the proof of Lemma 3.

We prove that in presence of a concurrent DLR.write() operation, a DLR.read() operation returns a blockchain that contains the block of the last valid DLR.write operation that happened before DLR.read() invocation, or the last block written by a valid DLR.write() operation concurrent with the DLR.read() operation. Let

```
Operation DLR_p.read () is % issued by any reader p%
(01)  wait Δ
(02)  return(best_chain())

Operation DLR_q.write (b) is % issued by writer q%
(03)  B = best_chain()
(04)  if update_tree(B,{b})
(05)     broadcast (<propose b>)
(06)     return true
(07)  else
(08)     return abort
-------------------------------------------------
(09)  upon deliver(<propose b>)
(10)     B = best_chain()
(11)     update_tree(B,{b})
```

Fig. 2. DLR.read() and DLR.write() operations of the SWMR Regular Distributed Ledger Register.

r be a DLR.read() operation issued by some process p such that r is concurrent with a valid DLR.write(b_1) operation w_1 and let w be the valid DLR.write(b) operation that precedes both w_1 and r. Two cases have to be analysed:

- Writer q is correct: Let t be the time at which operation w completes, and let t_1 be the time at which p invokes operation r. By the Δ-delivery property of the broadcast primitive, by time $t + \Delta$, all processes p have delivered b, and have updated their local copy DLR_p of the register. Thus at time $t_1 + \Delta > t + \Delta$ when p invokes the best_chain() function, p returns a blockchain that contains block b. Now suppose that q invokes w_1 at time $t_2 > t$. Since q is correct, then b_1 cryptographically links to b. Thus even if p receives b_1 before b, then the update_tree() function will first update DLR_p with b and then with b_1. Thus, at time $t_1 + \Delta$, operation r will returns a blockchain that either contains b or contains b_1.
- Writer q is a Byzantine process: The only way q may thwart the implementation is by invoking DLR$_q$.write() operations with non valid blocks, which is handled by the update_tree() function. This completes the proof of the lemma.

5.5 Atomic MWMR Distributed Ledger Register

An *atomic* MWMR distributed ledger register is a regular distributed ledger register that verifies the no new/old inversion property defined as follows:

- *no new/old inversion:* For any two DLR.read() operations r_1 and r_2 such that r_1 happens before $r2$ then the distributed ledger returned by $r2$ has the distributed ledger returned by r_1 as prefix.

In the following we first propose an implementation of the SWMR Atomic Distributed Ledger Register (see Figure 3), and then an implementation of the MWMR Atomic Distributed Ledger Register, that is a DLR register that can be DLR.written (and DLR.read) by multiple processes (see Figure 4).

```
Operation DLR_p.read () is % issued by any reader p%
(01)  wait Δ
(02)  return(best_chain() )

Operation DLR_q.write (b) is % issued by writer q%
(03)  B = best_chain()
(04)  if update_tree(B,{b})
(05)     broadcast (<propose b>)
(06)     wait Δ
(07)     return true
(08)  else
(09)     return abort
-------------------------------------------------
(10)  upon deliver(<propose b>)
(11)     B = best_chain()
(12)     if (b ∉ B ∧ update_tree (B, {b})) then
(13)        broadcast (<propose b>)
```

Fig. 3. DLR.read() and DLR.write() operations of the SWMR Atomic DLR register.

Lemma 5. *The implementation presented in Figure 3 implements a SWMR Atomic Distributed Ledger Register.*

Proof. Using a similar argument as Lemma 4 it trivially follows that Algorithm 3 satisfies the atomic SWMR DLR specification.

Consider two DLR.read() operations r_1 and r_2 such that r_1 happens before r_2. Let w be the last DLR.write() operation before r_1, and w_1 be a DLR.write() operation concurrent with both r_1 and r_2.

- Writer q is correct: Let b be the block broadcast by w_1 at t_1. Assume that the process that invoked r_1 delivered b_1 and returned as best chain the chain having b_1 as last block. Let r_2 be a read operation invoked by process p_2 such that r_2 happens after r_1. Necessarily, p_2 has delivered b_1 before returning its best chain since r_2 returns Δ time units after r_1 finishes. Since r_1 returned b_1 it follows that b_1 has been broadcast and by Δ-delivery property b_1 will be necessarily delivered by any correct process (including p_2) within Δ time. Let us now prove that if r_2 returns b then r_1 also returns b. Assume that r_1 returns b_1. By the above argument it follows that r_2 necessarily returns b_1 which contradicts the assumption that r_2 returns b.

– Writer q is a Byzantine process. Suppose that q can produce and broadcast a sequence of valid blocks during operation w_1. Let p_1 be the process that invokes r_1 and p_2 be the process that invokes r_2. Each time when q broadcasts a new valid block both p_1 and p_2 deliver it within Δ time. Let b_1 and b_2 be two valid blocks produced and broadcast by q and assume b_2 is cryptographically linked to b_1. Recall that the invalid blocks are discarded.
In the following we prove that r_2 returns a blockchain B_2 that has B_1 (the blockchain returned by r_1) as prefix. Assume by contradiction that b_1 is discarded by p_2. That is, p_2 updated its local blockchain with b_2 and discarded b_1. This is impossible since b_1 and b_2 are cryptographically linked. Assume now that b_1 and b_2 are not cryptographically linked and are produced such that both are cryptographically linked to the last block written by w. Since w happens before r_1 and r_2 both of them will have as last block in their best chain the block produced by w. Without restraining the generality assume that r_1 and r_2 have the same best chain before w_1 starts to broadcast b_1 and b_2. If b_1 or b_2 are delivered by p_1 then, since p_1 is broadcasting and r_2 waits Δ before returning the best chain, it follows that p_2 delivered b_1 and/or b_2 before returning its best chain.
It follows that the chain returned by r_2 has the chain returned by r_1 as prefix.

Hence, the implementation presented in Figure 3 satisfies no new/old inversion property.

```
Operation DLR_p.read () is % issued by any reader p %
(01)  wait Δ
(02)  return(best_chain() )

Operation DLR_p.write (b) is % issued by any writer p%
(03)  B = DLR.read ()
(04)  if update_tree(B,{b})
(05)     broadcast (<propose b>)
(06)     wait Δ
(07)     return true
(08)  else
(09)     return abort
-------------------------------------------------
(10)  upon deliver(<propose b>)
(11)     B = best_chain()
(12)     if (b ∉ B ∧ update_tree (B, {b})) then
(13)        broadcast (<propose b>)
```

Fig. 4. DLR.read() and DLR.write() operations of the MWMR Atomic DLR register.

Lemma 6. *The implementation presented in Figure 4 implements a MWMR Atomic Distributed Ledger Register.*

Proof. By Lemma 5 Algorithm 4 verifies the SWMR atomic register specification since it behaves exactly as Algorithm 3 except that the write operation starts with a read operaion. Note that it has no impact in a single writer setting. Consider two concurrent DLR.write() operations w_1 and w_2 invoked by processes p_1 and p_2. Let w be the last DLR.write() operation that happens before both w_1 and w_2 (i.e., no DLR.write() operation happens between w and w_1 and w_2). Let b be the block written by w. Let b_1 be the block broadcast by w_1 and b_2 be the block broadcast by w_2. By definition, both w_1 and w_2 update their local tree with the blocks b_1 and b_2 and wait Δ time units before returning. Let r be a read that happens after w_1 and w_2. The process p that issues r waits Δ before returning. Hence, p updates its local tree with both b_1 and b_2 and returns the best chain.

6 Distributed Ledger Register and Permissionless Blockchain Systems

In this section, we refine the specification of the Distributed Ledger Register proposed in [2] that mimics the behaviour of the Bitcoin and Ethereum distributed ledger (i.e., Bitcoin blockchain). In terms of read and write operations, the blockchain protocol informally translates as follows: when a miner wishes to create a new block, it first invokes a read operation on tree data structure managed by the miners, $\mathcal{TB}$. This read returns the longest chain of $\mathcal{TB}$, denoted by $\mathcal{B}$. From $\mathcal{B}$, the miner creates its new block, appends it to $\mathcal{B}$, and broadcasts $\mathcal{B}$ in the system (see [10] for details). Note that from a practical point of view, only the new block is broadcast to the system, and if necessary miners wait from their neighbours for blocks in $\mathcal{B}$ they are not aware of.

As recalled in Section 2, the level of confirmation k of a block b in a blockchain provides guarantees on the likelihood that b can be pruned from the blockchain. The blockchain properties are closely related to the value of k. Therefore, in [2] is introduced the notion of k-valid write. Definition below is just a refinement of the definition proposed in [2] considering that write operation is invoked with a block as parameter and not a chain.

Definition 1 (k-valid write). *Operation DLR.write(b) is k-valid if and only if there exist a time $t > 0$ and an integer $k > 0$ such that a DLR.read() invoked at time $t' > t$ after the invocation of DLR.write(b) returns a chain $\mathcal{B}'$ such that $\exists \mathcal{B}$ prefix of $\mathcal{B}'$ and $\mathsf{length}(\mathcal{B}') \geq \mathsf{length}(\mathcal{B}) + k$ and the last block of $\mathcal{B}$ is b, where function length$(\mathcal{B})$ returns the number of blocks that compose chain $\mathcal{B}$.*

Operation write(b) returns *true* if write(b) is k-valid otherwise it returns *abort*.

The presence of the genesis block is very similar to the classical assumption in registers theory which states that before the first read at least one virtual write operation happened. Therefore, for the distributed ledger register we consider that before the first read there was at least a virtual k-valid write.

```
Operation DLR.read () is % issued by a reader %
(01)  wait Δ
(02)  return(best_chain() )

Operation DLR.write (b) is % issued by a writer %
(03)  B = DLR.read()
(04)  if update_tree(B,{b})
(05)     broadcast (<propose b>)
(06)     wait Δ
(07)     repeat
(08)        B' = DLR.read ()
(09)     until length(B') ≥ length(B) + k
(10)     if  B= prefix(B')
(11)        return true
(12)  return abort
_____________________________________
_____________________________________
(13)  upon deliver(<propose b>)
(14)     B = best_chain()
(15)     if (b ∉ B ∧ update_tree (B, {b})) then
(16)        broadcast (<propose b>)
```

Fig. 5. DLR.read() and DLR.write() operations of the MWMR Atomic Distributed Ledger Register satisfying k-consistency property.

A ledger multi-reader multi-writer register defined in [2] that mimics Bitcoin/Ethereum has the following specification.

- **Liveness** Any invocation of a DLR.write(b) or a DLR.read() terminates.
- **k-consistency** Any DLR.read() returns a value $\mathcal{B}$ such that $\exists \mathcal{B}'$ prefix of $\mathcal{B}$ with last($\mathcal{B}'$) is the value of the register written by the last k-valid DLR.write() operation that precedes DLR.read().

In the following we propose to extend the implementation of an atomic DLR register in order to verify the **k-consistency** property specified above.

Lemma 7. *Algorithm 5 verifies the specification of SWMR Atomic Distributed Ledger Registers.*

Proof. In the sequel we assume that the writer is a correct. Consider an execution of Algorithm 5 such that any DLR.write() operation is followed by at least k DLR.write() operations. Let w_1 and r_1 be respectively DLR.write() and DLR.read() operations such that r_1 happens after w_1 and no DLR.write() operation is concurrent with r_1. Let b be the block broadcasted by w_1. Since r_1 happens after w_1 and r_1 returns only after Δ time units and the propagation of b takes at most Δ time units then r_1 returns a blockchain B that has b as last block. We now prove that in any execution of Algorithm 5 there is no new/old inversion. Consider two

DLR.read() operations r_1 and r_2 such that r_1 happens before r_2. Let w be the DLR.write() operation before r_1 and w_1 be a concurrent operation with both r_1 and r_2. Assume r_1 received the block broadcast during w_1. Let b be this block. Since r_2 happens after r_1 (hence Δ time units after the broadcast of b by r_1), then r_2 received b. Hence, Algorithm 5 verifies no new/old inversion property. The case of an incorrect writer is similar with the proofs of the previous section.

Lemma 8. *Algorithm 5 verifies the specification of MWMR Atomic Distributed Ledger Registers.*

Proof. By Lemma 7 Algorithm 5 verifies the SWMR Atomic Distributed Ledger Registers specification. Consider two concurrent DLR.write() operations w_1 and w_2. Let b be the block DLR.written by the DLR.write() operation that happens before w_1 and w_2. Let b_1 be the block broadcast by w_1 and, b_2 be the block broadcast by w_2. b_1 and b_2 are cryptographically linked to b either directly or indirectly. Both w_1 and w_2 after broadcasting enter the repeat loop and invoke a DLR.read() operation. Hence, both w_1 and w_2 wait Δ time units (the time necessary for b_1 and b_2 to be broadcast in the network). After Δ time units all nodes in the network have b_1 and b_2 in their local blockchain. Let $\prec$ be the total order defined by the function *best_chain*. Hence, either $b_1 \prec_b b_2$ or $b_2 \prec_b b_1$. Assume without loosing the generality that $b_1 \prec_b b_2$. Let r be a DLR.read() operation that happens after w_1 and w_2 and no concurrent write executes. r will return the blockchain having b_1 as last block.

Lemma 9. *Algorithm 5 verifies the k-consistency property.*

Proof. Let r be a DLR.read() operation that happens after the last valid DLR.write() operation w. Let b be the block broadcasted by w. Since w finished without abort then, by Lemma 7 r returns B' having b in its suffix.

7 Conclusions and Future directions

Recently several academic studies have been devoted to the formal specification of distributed ledgers [2–5]. Our work continues this effort and extends the previous work in several ways. Our work is the first one to propose a specification of distributed ledger register that matches the Lamport hierarchy [17] from safe to atomic registers. We have proposed new algorithms to implement distributed ledger registers with safe, regular and atomic guarantees on top of a broadcast primitive specific to distributed ledgers in presence of Byzantine processes. It should be noted that our work is complementary to the work proposed in [5] that focuses only on ledger objects that consist in a totally ordered sequence of blocks (or records). Unifying our framework with the one proposed in [5] and extending it to multi-objects operations [4] is an interesting open direction. Moreover, connecting this new framework with the runtime specification proposed in [3] in order to automatically design and verify distributed ledgers algorithms with various semantics is an interesting and important open research direction.

References

1. Marcos K. Aguilera, Svend Frolund, Vassos Hadzilacos, Stephanie L. Horn, and Sam Toueg. Abortable and query-abortable objects and their efficient implementation. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2007.
2. E. Anceaume, R. Ludinard, M. Potop-Butucaru, and F. Tronel. Bitcoin a distributed shared register. In *Proceedings of the International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, 2017.
3. E. Anceaume, A. Del Pozzo, R. Ludinard, M. Potop-Butucaru, and S. Tucci Piergiovanni. Blockchain abstract data type. In *Proceedings of the 31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2019.
4. A. F. Anta, C. Georgiou, and N. C. Nicolaou. Atomic appends: Selling cars and coordinating armies with multiple distributed ledgers. *Tokenomics 2019*, abs/1812.08446, 2018.
5. A. F. Anta, K. Konwar, C. Georgiou, and N. Nicolaou. Formalizing and implementing distributed ledger objects. *ACM SIGACT News*, 49(2):58–76, 2018.
6. C. Cachin. Blockchain - From the Anarchy of Cryptocurrencies to the Enterprise (Keynote Abstract). In *Proc. of the OPODIS International Conference*, 2016.
7. C. Decker, J. Seidel, and R. Wattenhofer. Bitcoin Meets Strong Consistency. In *Proc. of the ICDCN International Conference*, 2016.
8. C. Decker and R. Wattenhofer. Information propagation in the bitcoin network. In *13th IEEE International Conference on Peer-to-Peer Computing, IEEE P2P 2013, Trento, Italy, September 9-11, 2013, Proceedings*, pages 1–10, 2013.
9. I. Eyal, A. E. Gencer, E. G. Sirer, and R. Van Renesse. Bitcoin-NG: A scalable blockchain protocol. In *Procs of the USENIX NSDI Symposium*, 2016.
10. J. A. Garay, A. Kiayias, and N. Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Proc. of the EUROCRYPT International Conference*, 2015.
11. A. Girault, G. Gössler, R. Guerraoui, J. Hamza, and D.-A. Seredinschi. Monotonic prefix consistency in distributed systems. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, pages 41–57, Berlin, Germany, 2018. Springer.
12. M. Herlihy. Concurrency and availability as dual properties of replicated atomic data. *J. ACM*, 37(2):257–278, 1990.
13. M. Herlihy and M. Moir. Blockchains and the logic of accountability: Keynote address. In *Proc. of the ACM/IEEE LICS Symposium*, 2016.
14. J.-H. Hoepman, M. Papatriantafilou, and P. Tsigas. Self-stabilization of wait-free shared memory objects. *J. Parallel Distrib. Comput.*, 62(5):818–842, 2002.
15. G. O. Karame, E. Androulaki, M. Roeschlin, A. Gervais, and S. Čapkun. Misbehavior in Bitcoin: A Study of Double-Spending and Accountability. *ACM Trans. Inf. Syst. Secur.*, 18(1), 2015.
16. E. Kokoris-Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *Proc. of the USENIX Security Symposium*, 2016.
17. L. Lamport. On inter-process communications, part I: basic formalism and part II: algorithms. *Distributed Computing*, 1(2):77–101, 1986.
18. A. Miller and J. J. LaViola Jr. Anonymous byzantine consensus from moderately-hard puzzles: A model for bitcoin. http://bravenewcoin.com/assets/Whitepapers/, 2014.
19. S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

20. Nicolas C. Nicolaou, Antonio Fernández Anta, and Chryssis Georgiou. Cover-ability: Consistent versioning in asynchronous, fail-prone, message-passing environments. In *Proceedings of the IEEE International Symposium on Network Computing and Applications (NCA)*, 2016.
21. R. Pass and E. Shi. Fruitchains: A fair blockchain. In *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017*, pages 315–324, New York, NY, USA, 2017. ACM.
22. R. Pass R., L. Seeman, and A. Shelat. Analysis of the blockchain protocol in asynchronous networks. In *Proc. of the EUROCRYPT International Conference*, 2017.

Appendix: Background on Distributed Registers

This section recalls the main properties of classical distributed read-write registers.

A distributed read-write register REG is a shared variable accessed by a set of processes through two operations, namely REG.write() and REG.read(). Informally, the REG.write() operation updates the value stored in the shared variable while the REG.read() obtains the value contained in the shared variable. Every operation issued on a register is, generally, not instantaneous and can be characterised by two events occurring at its boundaries: an *invocation* event and a *reply* event. Both events occur at two different instants with respect to the fictional global time: the invocation event of an operation op (i.e., $op = REG$.write() or $op = REG$.read()) occurs at the invocation time denoted by $t_B(op)$ and the reply event of op occurs at the reply time denoted by $t_E(op)$.

Given two operations op and op' on a register, we say that op *precedes* op' ($op \prec op'$) if and only if $t_E(op) < t_B(op')$. If op does not precede op' and op' does not precede op, then op and op' are *concurrent* (noted $op||op'$).

An operation op is *terminated* if both the invocation event and the reply event occurred (*i.e.*, the process executing the operation does not crash between the invocation time and the reply time). We suppose that a terminated operation can either be successful and thus returns *true* or can return *abort* when, for example, some operational conditions are not met [1]. More details appear in [2]. On the other hand, an operation that does not terminate is called *failed*.

When read() and write() operations concurrently access a shared variable, the meaning of *last written value* becomes ambiguous. Depending on the semantics of the operations, three types of registers have been defined by Lamport [17], i.e., *safe*, *regular* and *atomic*. In the following we recall the specifications of these registers when processes are subject to Byzantine faults.

A *safe* single-writer multiple-reader, (SWMR) register REG is a distributed register satisfying the following two properties. Let p be any correct process.

- *Termination*: Any invocation of REG.write() or REG.read() by process p eventually terminates.
- *Validity*: A REG.read() operation invoked by p returns the last value written before its invocation (*i.e.* the value written by the latest REG.write() preceding this REG.read() operation), or any value of the register domain in case the REG.read() operation is concurrent to a REG.write() operation.

A *regular* single-writer multiple-reader, (SWMR) register REG is a distributed register satisfying the following two properties. Let p be a correct process.

- *Termination*: Any invocation of REG.write() or REG.read() by process p eventually terminates.
- *Validity*: A REG.read() operation invoked by p returns the last value written before its invocation (*i.e.* the value written by the latest REG.write() preceding this REG.read() operation), or a value written by a REG.write() operation concurrent with it.

An *atomic* single-writer multi-reader, (SWMR) distributed register REG is a regular SWMR register with the additional property. Let p and q be any two correct processes (where p is not necessarily distinct from q).

- *order*: Any two successive invocations of the REG.read() operation by p and q that overlap the same REG.write() operation cannot return the new and then the old value.

An *atomic* multi-writer, multi-reader, (MWMR) distributed register REG is a distributed register that satisfies the following three properties. Let p and q be any two correct processes (where p is not necessarily distinct from q).

- *Termination*: Any invocation of REG.write() or REG.read() by process p eventually terminates.
- *Validity*: A REG.read() operation invoked by p returns the last value written before its invocation (*i.e.* the value written by the latest REG.write() preceding this REG.read() operation), or a value written by a REG.write() operation concurrent with it.
- *Ordering*: If a REG.read() invoked by p returns a value v and a subsequent REG.read() invoked by q returns a value w, then the REG.write() of w does not precede the REG.write() of v.

Roughly speaking, the ordering property prevents an "old" value from being read by process p once a "newer" value has been read by process q.