HAL
open science

Distributed Ledger Register: From Safe to Atomic

Emmanuelle Anceaume, Marina Papatriantafilou, Maria Potop-Butucaru, Philippas Tsigas

To cite this version:

Emmanuelle Anceaume, Marina Papatriantafilou, Maria Potop-Butucaru, Philippas Tsigas. Distributed Ledger Register: From Safe to Atomic. 2019. hal-02201472

HAL Id: hal-02201472

https://hal.science/hal-02201472v1

Preprint submitted on 31 Jul 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed Ledger Register: From Safe to Atomic

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

[1] CNRS, Université Rennes, Inria, IRISA, Rennes, France
[2] Dept. of Computer Science and Engineering (CSE), Chalmers University of Technology, Sweden
[3] LIP6, Sorbonne University, Paris, 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 guaranties in a model of communication specific to distributed ledgers technology that we also formalize. Then, we propose an implementation of a distributed ledger register that satisfies the atomic specification and the k-consistency property that characterises the permissionless distributed blockchains such as Bitcoin and Ethereum.

1 Introduction

The blockchain is probably one of the hottest and most transformative topic in the current digital landscape. The bitcoin crypto-currency [18], the leading application of the blockchain technology, has shown remarkably steady, 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 entrusted, 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. The point is that 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.

Such challenges can be mitigated by laying down the theoretical foundations of distributed ledgers. Links between the distributed computing theory and distributed ledger has been pioneered by Garay et *al* [11]. The main focus of the distributed community [7, 8, 10, 11, 14, 16, 21] 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 for making possible to 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 [13], 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 semantic.

Our contribution. This paper extends the work of [2] in several ways. First, we propose a specification of distributed ledger register that matches the Lamport hierarchy [17] from safe to atomic. Furthermore, we propose an implementation of the distributed ledger register that verify safe, regular and atomic semantics in a model of communication specific to distributed ledgers technology ([9]) where the system provides a broadcast primitive satisfying Δ-delivery (if a node invokes $\mathsf{broadcast}(m)$ then every correct node eventually $\mathsf{delivers}$ m within Δ time units). Then 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 4 presents the computational model. Section 5 provides a brief summary of shared registers and their specification. In Section 6 we specify the Distributed Ledger Register and propose implementations of Distributed Ledger Register satisfying safe, regular and atomic semantic. Section 7 makes the connection between the Distributed Ledger Register specification defined in Section 6 and distributed ledgers. Section 8 concludes and presents open problems.

2 Permissionless Blockchains

In 2008, Satoshi Nakamoto, a pseudonymous author, published a white paper describing the Bitcoin network, a way to create, distribute and manage a currency that does not rely on a trusted third party [18]. Since then many crypto-currencies have been proposed, including the popular Ethereum [22]. 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.

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 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 – 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*, a unique branch to represent the history of the crypto-system – this branch being the longest one of the tree, or equivalently the one that required the most important quantity of work. The best chain in Bitcoin corresponds to the longest chain starting from the genesis block of the distributed ledger (the blockchain is bootstrapped with the genesis block), while in Ethereum, the best chain is the heaviest one. The *level of confirmation* of a block b belonging to the best chain of the distributed ledger is equal to the number of blocks included in the best chain starting from b. We say that a transaction is *deeply confirmed* once it reaches such a confirmation level.

Garay and Kiayias [11] have characterized Bitcoin blockchain via its quality and its common prefix properties. Specifically, they have shown that, by keeping the creation rate of blocks very low with respect to their propagation time in the network *(i)* if the adversary controls no more than 1/3 of the network hashing power then it provably controls less than a majority of the blocks in the blockchain and, *(ii)* if the adversary controls no more than 1/2 of the network hashing power then the blockchains maintained by any two honest nodes possess a large common prefix (up to the last k appended blocks), and the probability that they are not mutual prefix of each other decays exponentially in k.

3 Related works

In [11] the authors extract Bitcoin backbone and define invariant that this protocol has to satisfy in order to verify with high probability an eventual consistent prefix. This line of work has been continued by [20]. However, to the best of our knowledge, no other previous attempt proposed a consistency unified framework and hierarchy capturing both Consensus-based and proof-of-work based blockchains. In [1], the authors present a study about the relationships between Byzantine fault tolerant consensus and distributed blockchains. In order to abstract out the proof-of-work mechanism the authors propose a specific oracle, in the same spirit of our oracle abstraction, but more specific than ours, since it makes a direct reference to proof-of-work properties. In parallel and independently of our work, [6] proposes 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 our results about the necessity of Consensus to solve Strong Prefix. In [12] the authors 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 [3], the authors 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. It is already known that some blockchain implementations solve eventual consistency of an append-only queue using Consensus [4, 6]. However, Bitcoin [19] and Ethereum [22] that technically do not solve Consensus need to be characterized from the point of view of the consistency guaranties that they are able to offer. The question is about the consistency criterion of blockchains as Bitcoin [19] and Ethereum [22] that technically do not solve Consensus, and their relation with Consensus in general. The first work that addressed this open question was proposed by [2]. They proposed to connect distributed registers theory to distributed ledgers. They introduce the notion of Distributed Ledger Register (DLR) were the value of the register has a tree topology instead of a single value as in the classical theory of distributed registers. The nodes of the tree are blocks of transactions cryptographically linked. This work has been further extended in [3]. They specify the consistency guarantees offered by various ledgers at runtime. 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 their mutual objective to contribute in the provision of a proper formulation of distributed ledgers in terms of shared objects, the authors of [6] open a line of work complementary to the one presented in [2] and the extension presented in the current paper. In [6] the ledger object is a totally ordered sequence of blocks (or records) while [2] and the current work consider a tree of blocks that specify both permissioned and permissionless ledgers. The work proposed in [6] was continued in [5] by defining Multi-Distributed Ledger Objects (MDLO), which is the result of aggregating multiple Distributed Ledger Objects satisfying the definition proposed in [6]. In Section 8 we will discuss the research directions opened by [2, 3, 5, 6] and the current work.

4 Computational Model

We model the system as a partially synchronous distributed system composed of an arbitrary finite number of nodes, such that each node has enough computation resources to mine blocks.

Each node 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 node in the system. The passage of time is measured by a fictional global clock. Nodes do not have access to the fictional global time. At each time t, each node is characterised by its local state.

It is assumed that the system has a built-in communication abstraction, denoted broadcast, that allows nodes to communicate by exchanging messages via $\mathsf{broadcast}()$ and $\mathsf{deliver}()$ operations. This communication abstraction is defined by the following properties.

- *Δ-delivery.* There exists $\Delta > 0$ such that if a node invokes $\mathsf{broadcast}(m)$ then every correct node eventually $\mathsf{delivers}$ m within Δ time units.
- *Validity.* If a correct node $\mathsf{delivers}$ a message m from p then p has previously invoked $\mathsf{broadcast}(m)$.

By *correct* node, we mean a node that follows the prescribed protocols. We suppose on the other hand that some of them can suffer arbitrary failures—such nodes are called *incorrect*. For instance, an incorrect node can manipulate the communication primitive by broadcasting inconsistent messages, or by not broadcasting messages or by stopping its execution. We assume that less than a third of the computational power of the system is owned by incorrect nodes. We do not make any restrictions regarding incorrect users.

5 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 $REG.\mathsf{write}()$ and $REG.\mathsf{read}()$.

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 $op||op'$). In case of concurrency while accessing the 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]: *safe*, *regular* and *atomic*.

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). A terminated operation can either be successful and thus returns *true* or can return *abort* when, for example, some operational conditions are not met. More details will be given in Section 7. On the other hand, an operation that does not terminate is called *failed*.

The semantic of a distributed read-write register (simply called read-write register) can be classified as *safe*, *regular* or *atomic* [17]. The *safe* register ensures that a read which does not overlap with a write returns the last completed write. The result of a read overlapping a write can be any value from the register domain. The *regular* register verifies the safe semantic when reads are not concurrent with writes. For reads concurrent with writes the read will return either the last written value or the value of the concurrent write. A *safe* distributed register REG is defined by the following properties:

- *Liveness*: Any invocation of REG.write() or REG.read() eventually terminates.
- *Safety*: A REG.read() operation 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* distributed register REG is defined by the following properties:

- *Liveness*: Any invocation of REG.write() or REG.read() eventually terminates.
- *Safety*: A REG.read() operation 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* register is a regular register that satisfies the no new/old inversion property defined as follows:

– *no new/old inversion:* For any two read operations, the set of writes that do not strictly follow either of them must be perceived by both reads as occurring in the same order.

6 From Classical Distributed Registers to Distributed Ledger Registers

Interestingly enough, neither safety, regularity nor atomicity of distributed shared registers capture 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.

Interestingly, blockchains behavior has some similarities with the stabilizing registers, line of research pioneered by [15]. Looking at the stabilizing register, it implements some type of eventual consistency, in the sense that, there exists a prefix of the system execution for which there are no guarantees on the value of the shared register: register semantics hold only from a certain time in the execution. In contrast, the prefix of the blockchain eventually converges at every entity, while no guarantees hold for the last created blocks.

In [2] we have proposed a new register called the *Distributed Ledger Register* (DLR). However, the specification proposed in [2] is tailored for Bitcoin and Ethereum as we further discuss in Section 7. In the following we refine this specification 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 is a sequence of blocks. The *value* of DLR is the best chain of the tree. The $\mathsf{best_chain}(\mathcal{TB})$ is a chain in the tree $\mathcal{TB}$ that satisfies some predicate: the longest sequence of blocks, starting from the root (as in the case of Bitcoin) or the heaviest chain the tree (as in the case of Ethereum). The value of the DLR is called a *blockchain* and is denoted by $\mathcal{B}$. The DLR is equipped with write and read operations. The DLR.write operation allows any process to append to the value of DLR a block b. The DLR.$\mathsf{read}()$ operation allows any miner to retrieve the value of DLR.

In the following, we present the implementations of the distributed ledger register that satisfy safe, regular and atomic semantics assuming the communication abstraction satisfies the Δ-delivery property (i.e. there exists $\Delta > 0$ such that if a process invokes *broadcast(m)* then every correct process eventually delivers m within Δ time units.)

6.1 Safe Distributed Ledger Register

We propose an implementation of the Single Writer Multiple Reader (SWMR) Distributed Ledger Register that respects the safe semantic.

A *safe* distributed ledger register *DLR* is defined by the following properties:

- *Liveness*: Any invocation of DLR.write() or DLR.read() eventually terminates.
- *Safety*: A DLR.read() operation returns a chain $\mathcal{B}$ such that $\mathcal{B}$ has a prefix $\mathcal{B}_1$ whose last($\mathcal{B}_1$) is the last value DLR.written before its invocation (*i.e.* the value DLR.written by the latest DLR.write() preceding this DLR.read() operation), or any value of the register domain in case the DLR.read() operation is concurrent to a DLR.write() operation. Note that in the case of a DLR register a value in the register domain may be any distributed ledger.

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

Operation DLR.write (b) is % issued by a writer %
(02)  B = DLR.read()
(03)  append b to B
(04)  broadcast (<propose B>)
(05)  update_tree(TB, B)
(06)  wait Δ
(07)  return true
------------------------------------------
(08)  upon deliver(<propose B>)
(09)     update_tree(TB, B)
```

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

Lemma 1. *Algorithm 1 implements a SWMR Safe Distributed Ledger Register.*

Proof. Let r be a DLR.read() operation issued by a process p. Let w be the last DLR.write before r and no concurrent DLR.write is executed with r. Let $\mathcal{B}$ be the chain broadcast by w. Since r starts after w returns and w waits Δ before returning, then p receives $\mathcal{B}$, and returns it.

6.2 Regular Distributed Ledger Register

A *regular* distributed ledger register DLR is defined by the following properties:

- *Liveness*: Any invocation of DLR.write() or DLR.read() eventually terminates.
- *Safety*: A DLR.read() operation returns a chain $\mathcal{B}$ such that $\mathcal{B}$ has a prefix $\mathcal{B}_1$ whose last($\mathcal{B}_1$) is the last value DLR.written before its invocation (*i.e.* the value DLR.written by the latest DLR.write() preceding this DLR.read() operation), or a value DLR.written by a DLR.write() operation concurrent with it.

We propose an implementation of a SWMR Distributed Ledger Register that satisfies the regular semantic.

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

Operation DLR.write (b) is % issued by a writer %
(03) B = DLR.read()
(04) append b to B
(05) broadcast (<propose B>)
(06) update_tree(TB, B)
(07) return true
---------------------------------------
(08) upon deliver(<propose B>)
(09)    update_tree(TB, B)
```

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

Lemma 2. *Algorithm 2 implements a SWMR Regular Distributed Ledger Register.*

Proof. Algorithm 2 implements a safe register. Let r be a DLR.read() operation issued by process p such that r happened after some DLR.write() operation w. Let $\mathcal{B}$ be the chain broadcast during w. Since p waits Δ time units before returning then p received $\mathcal{B}$ before returning. Hence p returns the last DLR.written value.

We now prove that in presence of a concurrent DLR.write() operation, a DLR.read() operation returns either the concurrent DLR.written value or the previous one. Let r be a DLR.read() operation issued by some process p such that r is concurrent with a DLR.write() operation w_1 and let w be the last DLR.write() operation that precedes both w_1 and r. Let $\mathcal{B}$ be the chain broadcast by w. Since w_1 starts with a DLR.read() operation then chain $\mathcal{B}_1$ broadcast by w_1 has $\mathcal{B}$ as prefix (by the safe property of the register). Operation r receives $\mathcal{B}$ (since r happened after w), hence the tree structure maintained by p has been updated with $\mathcal{B}$. If p received $\mathcal{B}_1$ then p updated its tree with $\mathcal{B}_1$ hence its best chain now is $\mathcal{B}_1$. Hence, p returns either $\mathcal{B}$ or $\mathcal{B}_1$ which are the result of the last DLR.write() or a concurrent DLR.write() operation.

6.3 Atomic Distributed Ledger Register

An *atomic* 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, 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.

```
Operation DLR.read () is % issued by a reader %
(01) broadcast (<propose best_chain(TB) >)
(02) wait Δ
(03) return(best_chain(TB) )

Operation DLR.write (b) is % issued by a writer %
(04) B = DLR.read()
(05) append b to B
(06) broadcast (<propose B>)
(07) update_tree(TB, B)
(08) return true
-------------------------------------
(09) upon deliver(<propose B>)
(10)    update_tree(TB, B)
```

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

Lemma 3. *Algorithm 3 implements a SWMR Atomic Distributed Ledger Register.*

Proof. From Lemma 2 it trivially follows that Algorithm 3 satisfies the SWMR Regular 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. Assume r_1 received the distributed ledger $\mathcal{B}$ broadcast during w_1. Since r_2 happens after r_1 (hence after Δ time units after the broadcast of $\mathcal{B}$), then r_2 received $\mathcal{B}$. Hence, Algorithm 3 satisfies no new/old inversion property.

Lemma 4. *Algorithm 4 implements a MWMR Atomic Distributed Ledger Register.*

Proof. By Lemma 3 Algorithm 4 verifies the SWMR atomic register specification since it behaves exactly as Algorithm 3 except that the "wait Δ" line (line 8) has been added. Note that it has no impact in a single writer setting. Consider two concurrent *DLR*.write() operations w_1 and w_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 chain written by w. Let B_1 be the chain broadcast by w_1 and B_2 be the chain broadcast by w_2. By definition B is prefix of both B_1 and B_2. Both w_1 and w_2 update their local tree with the chains B_1 and B_2 and wait Δ time units before returning. Let r be a read that

```
Operation DLR.read () is % issued by a reader %
(01) broadcast (<propose best_chain(TB) >)
(02) wait Δ
(03) return(best_chain(TB) )

Operation DLR.write (b) is % issued by a writer %
(04) B = DLR.read()
(05) append b to B
(06) broadcast (<propose B>)
(07) update_tree(TB, B)
(08) wait Δ
(09) return true
-------------------------------------------
(10) upon deliver(<propose B>)
(11)    update_tree(TB, B)
```

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

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 between B_1 and B_2.

7 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). We recall the functioning of Bitcoin. As described in the introduction, each miner needs to locally manage a data structure from which it can extract the blockchain. Specifically, this data structure is a tree, denoted by $\mathcal{TB}$, and the blockchain, denoted by $\mathcal{B}$, is the longest chain in this tree. By construction, the root of $\mathcal{TB}$ is the genesis block, a common block for all the miners. 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 $\mathcal{TB}$. 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 [11] 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 virtual 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*. As described in Section 2 the value of k depends on the proportion β of malicious miners in the system. It has been shown by Nakamoto [18], that if the proportion β of malicious miners is $\leq 10\%$, then with probability $\leq 0.1\%$, a transaction can be rejected if its level of confirmation in a local copy of the blockchain is less than or equal to than 6.

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.

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

Proof. Consider an execution of Algorithm 5 such that any $\mathsf{DLR.write}()$ operation is followed by at least k $\mathsf{DLR.write}()$ operations. Let w_1 and r_1 be respectively a $\mathsf{DLR.write}()$ and $\mathsf{DLR.read}()$ operations such that r_1 happens after w_1 and no $\mathsf{DLR.write}()$ operation is concurrent with r_1. Let B be the chain broadcast 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 B. We now prove that in any execution of Algorithm 5 there is no new/old inversion. Consider two $\mathsf{DLR.read}()$ operations r_1 and r_2 such that r_1 happens before r_2. Let w be the $\mathsf{DLR.write}()$ operation before r_1 and w_1 be a concurrent operation with both r_1 and r_2. Assume r_1 received the blockchain broadcast during w_1. Let $\mathcal{B}$ be this blockchain. Since r_2 happens after r_1 (hence Δ time units after the broadcast of $\mathcal{B}$ by r_1), then r_2 received $\mathcal{B}$. Hence, Algorithm 5 verifies no new/old inversion property.

Lemma 6. *Algorithm 5 verifies the specification of MWMR Atomic Distributed Ledger Registers.*

```
Operation DLR.read () is % issued by a reader %
(01) broadcast (<propose best_chain(TB) >)
(02) wait Δ
(03) return(best_chain(TB) )

Operation DLR.write (b) is % issued by a writer %
(04) B = DLR.read()
(05) append b to B
(06) broadcast (<propose B>)
(07) update_tree(TB, B)
(08) repeat
(09)    B' = DLR.read ()
(10) until length(B') ≥ length(B) + k
(11) if  B= prefix(B')
(12)    return true
(13) else return abort
-------------------------------------
(14) upon deliver(<propose B>)
(15)    update_tree(TB, B)
```

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

Proof. By Lemma 5 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 chain DLR.written by the DLR.write() operation that happens before w_1 and w_2. Let B_1 be the chain broadcast by w_1 and, B_2 be the chain broadcast by w_2. B_1 and B_2 have both B as prefix. 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 B_1.

Lemma 7. *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 the blockchain broadcast by w. Since w finished without abort then, by Lemma 5 r returns B' with B prefix of B'.

8 Conclusions and Future directions

Recently several academic studies have been devoted to the formal specification of distributed ledgers [2, 3, 5, 6]. 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 [9]. We have also formalized this primitive. Then, we have proposed an implementation of a distributed ledger register that satisfies the atomic specification and the k-consistency property that characterizes the permissionless distributed blockchains such as Bitcoin and Ethereum as proved in [2]. Therefore, we respond to an open problem raised by [2] where it has been proven that Bitcoin and Ethereum implement a distributed ledger register verifying only regular semantics. It should be noted that our work is complementary to the work proposed in [6] 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 [6] and extending it to multi-objects operations [5] 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. I. Abraham and D. Malkhi. The blockchain consensus layer and BFT. *Bulletin of the EATCS*, 3(123):1–23, 2017.
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. E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolic, S. Weed Cocco, and J. Yellick. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. https://arxiv.org/pdf/1801.10228v1.pdf, 2018.
5. 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. URL: http://arxiv.org/abs/1812.08446.
6. A. F. Anta, K. Konwar, C. Georgiou, and N. Nicolaou. Formalizing and implementing distributed ledger objects. *ACM SIGACT News*, 49(2):58–76, 2018.
7. C. Cachin. Blockchain - From the Anarchy of Cryptocurrencies to the Enterprise (Keynote Abstract). In *Proc. of the OPODIS International Conference*, 2016.
8. C. Decker, J. Seidel, and R. Wattenhofer. Bitcoin Meets Strong Consistency. In *Proc. of the ICDCN International Conference*, 2016.
9. 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.

10. 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.
11. J. A. Garay, A. Kiayias, and N. Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Proc. of the EUROCRYPT International Conference*, 2015.
12. 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.
13. M. Herlihy. Concurrency and availability as dual properties of replicated atomic data. *J. ACM*, 37(2):257–278, 1990.
14. M. Herlihy and M. Moir. Blockchains and the logic of accountability: Keynote address. In *Proc. of the ACM/IEEE LICS Symposium*, 2016.
15. 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.
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. S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.
19. S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/bitcoin.pdf, 2008.
20. 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.
21. R. Pass R., L. Seeman, and A. Shelat. Analysis of the blockchain protocol in asynchronous networks. In *Proc. of the EUROCRYPT International Conference*, 2017.
22. G. Wood. Ethereum: A secure decentralised generalised transaction ledger. http://gavwood.com/Paper.pdf.