Bitcoin a Distributed Shared Register

Emmanuelle Anceaume[1], Romaric Ludinard[2], Maria Potop-Butucaru[3], and Frédéric Tronel[4]

[1] CNRS / IRISA, Campus de beaulieu, Rennes, France
[2] CREST / ENSAI, Rennes, France
[3] LIP6, Université P. & M. Curie, Paris, France
[4] CentraleSupélec, Rennes, France

Abstract. Distributed Ledgers (e.g. Bitcoin) occupy currently the first lines of the economical and political media and many speculations are done with respect to their level of coherence and their computability power. Interestingly, there is no consensus on the properties and abstractions that fully capture the behaviour of distributed ledgers. The interest in formalising the behaviour of distributed ledgers is twofold. Firstly, it helps to prove the correctness of the algorithms that implement existing distributed ledgers and explore their limits with respect to an unfriendly environment and target applications. Secondly, it facilitates the identification of the minimal building blocks necessary to implement the distributed ledger in a specific environment.
Even though the behaviour of distributed ledgers is similar to abstractions that have been deeply studied for decades in distributed systems no abstraction is sufficiently powerful to capture the distributed ledger behaviour.
This paper introduces the Distributed Ledger Register, a register that mimics the behaviour of one of the most popular distributed ledger, i.e. the Bitcoin ledger. The aim of our work is to provide formal guarantees on the coherent evolution of Bitcoin. We furthermore show that the Bitcoin blockchain maintenance algorithm verifies the distributed ledger register properties under strict conditions. Moreover, we prove that the Distributed Ledger Register verifies the regularity register specification. It follows that the strongest coherency implemented by Bitcoin is regularity under strong assumptions (i.e. partial synchronous systems and sparse reads). This study contradicts the common belief that Bitcoin implements strong coherency criteria in a totally asynchronous system. To the best of our knowledge, our work is the first one that makes the connection between the distributed ledgers and the classical theory of distributed shared registers.

1 Introduction

Blockchain has become one of the most omnipresent buzzwords in economical, political and scientific media. Bitcoin [16] and Ethereum [19], the most popular blockchain applications nowadays are cited as the universal solution for managing a broad range of goods ranging bank accounts and client transactions operations to energy or notarial agreements management. Political analysts predict that blockchains will be used in the near future as regular bases in administration or national and international economical exchanges.

Bitcoin and Ethereum, beyond their incontestable assets such as decentralisation, simple design and relative easy use, are neither riskiness nor limitations free. For example, the most popular issue that has been reported regarding Ethereum functioning was the theft of 60 million dollars due to the exploitation of an error in a smart contract code. It seems clear that neither Bitcoin nor Ethereum are mature enough to be used in critical economical and administrative applications, as shown by a recent scientific analysis [6] which enlightens the main limitations exposed by Bitcoin, including low quality of services, storage limitations, low throughput, high cost, security weakness, and weak coherency. The point is that an increasing number of areas promote the use of blockchains for the development of their applications, and undeniably, the properties enjoyed by these blockchains should be studied to fit the applications requirements, together with their relationships with blockchain-based applications.

Such challenges can be mitigated by laying down the theoretical foundations of blockchains, and more generally distributed ledgers. Connection between the distributed computing theory and Bitcoin distributed ledger has been pioneered by Garay et *al* [10]. The main focus of the distributed community [5,7,9–11,13,18] has so far been the distributed ledger agreement aspects. Our paper investigates consistency properties of the distributed ledger and tries to make the connection between the distributed ledgers and the distributed registers theory.

Our contribution. Interestingly, the Bitcoin related literature is not yet agreeing on the level of coherency offered by Bitcoin. Some of the studies, as for example the one carried off by Decker et al [8] advocate for strong consistency. Before discussing the level of consistency verified by Bitcoin one should first capture the properties of this system in terms of safety and liveness. To the best of our knowledge, none of the previous cited works formalised the consistency properties offered by the Bitcoin blockchain. The aim of our work is to provide formal guarantees on the coherent evolution of Bitcoin. Our work is the first one that makes the connection between the distributed ledgers and the classical theory of distributed shared registers. First, we show that the classical definitions of registers including their stabilisation extensions do not capture Bitcoin behaviour. Then, we formalise the Distributed Ledger Register that mimics the behaviour of Bitcoin. We finally show that the Bitcoin blockchain maintenance algorithm verifies the distributed ledger register properties.

Paper Roadmap The remaining of the paper is organised as follows. Section 2 recalls the main principles of the Bitcoin system, and Section 3 presents its computational model. Section 4 provides a brief summary of shared registers and their extensions. We end this section by enlightening why these definitions do not fully captures the Bitcoin behaviour. In Section 5, we extend the registers theory with a new register that we call the Distributed Ledger Register, and we show that Bitcoin implements such a register. Section 6 concludes and presents some open problems.

2 Bitcoin Background

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 [17]. Since then many crypto-currencies have been proposed, including the popular Ethereum [19]. An implementation of Bitcoin was released shortly after under the name Bitcoin Core. In the following we focus on the functioning of Bitcoin, since Ethereum follows almost the same pattern and its differences are not relevant for our study. Most of the following is drawn from [3].

The Bitcoin network is a peer-to-peer payment network that relies on distributed algorithms and cryptographic functions to allow entities to pseudonymously buy goods with digital currencies called bitcoins. Bitcoin mainly relies on three types of data structures (i.e transactions, blocks and the distributed ledger – also called the blockchain) and three types of entities (i.e., user, Bitcoin node and miner) to offer such functionalities.

Transactions allow *users* to transfer bitcoins from a set of input accounts to a set of output accounts. An account is described by a key, derived from the public key of the public/private key generated by Bitcoin users. Note that to hide their profile, users should generate a new public/private key for each transaction they are recipient of. Keys are used to prove the ownership of bitcoins. Recipients of a transaction are credited once the transaction is confirmed in the blockchain. Users voluntarily pay a small *transaction fee* which will be kept by the miner that will succeed in confirming users transaction in the blockchain. In this case, the total amount of bitcoins in the input accounts is greater than the amount of bitcoins transferred to the output accounts.

To describe the evolution of user accounts, Anceaume et al [3] have adopted a place/transition model as depicted in Figure 1. User accounts are represented by places (circles) and transactions by transitions (vertical bars). The place from which an arc runs to a transition is an input place of the transition, and the place to which an arc runs to, is an output place of the transition. The number of bitcoins in a user account represents the tokens of the place. A transition may fire if there are sufficiently many tokens in its input places, and it consumes all of them upon firing. Places and transitions are dynamically created. In Figure 1, Alice creates transaction T_1 to transfer the 50 bitcoins of her account a_1 to Bob and Carol's accounts: 30 bitcoins to b_1 and 20 to c_1. Transaction T_4 contains a transaction fee equal to $(25 + 20) - (20 + 21 + 3) = 1$ bitcoin. Transaction T_2 is a special transaction called *coinbase*. Coinbase transactions are the way bitcoins are created, and their amount is currently set to 12.5 bitcoins plus the transaction fees included in the block (more details are given in the sequel).

A transaction T is locally valid at Bitcoin node p if p has received all the transactions that have credited all the input accounts of T and has never received

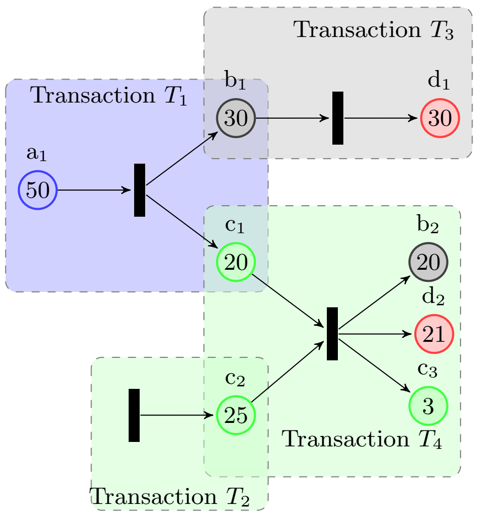

Fig. 1. Modelling the evolution of users' accounts

transactions already using any of those inputs. Indeed, an important aspect of Bitcoin accounts is their indivisibly, meaning that once an account has been created by a user, it will be credited by a single transaction and will be debited by a single subsequent transaction. If there exists some transaction T' such that both T and T' share some input account, then this input account is said to be in a double-spending situation. We say that transaction T is *conflict-free* if none of the input accounts of T is involved in a double-spending situation and all of the transactions that credited T's inputs are conflict-free. By construction, the induction is finite because Bitcoin creates money only through coinbase transactions, which do not rely on input accounts.

The solution adopted in Bitcoin to mitigate double-spending attacks, without relying on a central trusted authority, consists in gathering transactions into blocks and totally ordering them in a publicly accessible and distributively managed ledger. This is the role of *miners*.

A *block* contains a list of transactions, a reference to its parent block (hence the name of *blockchain*), and a proof-of-work, that is a nonce such that the hash of the block matches a given target. This target is calibrated so that the average generation time of a block by the network is equal to 10 minutes despite fluctuations of the peer-to-peer network.

We say that a block b is locally valid if it only contains locally valid transactions. *Bitcoin nodes* locally maintain a copy of the blockchain, and once validated, propagate newly transactions and blocks to all the entities of Bitcoin. Blocks are generated by *miners*, a subset of the Bitcoin nodes involved in the *proof-of-work* competition. The incentive to participate to such a competition is provided by the coinbase transactions that are credited to the successful miner accounts. This competition may result in multiple blocks referencing the very same parent block, and hence the creation of a tree with several chains. This situation is known as *blockchain fork*. Bitcoin defines the notion of *best chain* (the common history of the distributed ledger on which all Bitcoin nodes agree), which corresponds to the longest chain starting from the genesis block of the distributed ledger (the blockchain is bootstrapped with the genesis block). In the case of 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. Nakamoto [17] as well as subsequent studies [10,12,15] has shown that if the proportion 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 6. In case of Ethereum this level is not well defined, and seems to be around 12 [1]. We say that a transaction is *deeply confirmed* once it reaches such a confirmation level.

3 Computing model

We model the Bitcoin system as a partially synchronous distributed system (Distributed Ledger system) composed of an arbitrary finite number of users, miners and bitcoin nodes. In the following we assume that all bitcoin nodes have enough computation resources to mine blocks. Thus we do not distinguish anymore miners from bitcoin nodes.

Each miner in the distributed system is a state machine, whose state, called “local state”, is defined by the current values of its local variables. A configuration, or global state, of the Distributed Ledger system is composed of the local state of each miner in the system. The passage of time is measured by a fictional global clock. Miners do not have access to the fictional global time. At each time t, each miner is characterised by its local state.

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

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

By correct miner, we mean a miner that follows the prescribed protocols. We suppose on the other hand that some of them can suffer arbitrary failures—such miners are said incorrect. For instance, an incorrect miner 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 miners. No such restrictions hold for incorrect users.

4 Background on Distributed Registers

This section recalls the main properties of classical distributed registers, and shows that with these definitions, we cannot entirely describe the properties of the blockchain. Hence the need for a new type of register.

A distributed register is a shared variable accessed by a set of processes through two operations, namely $REG.\mathsf{write}()$ and $REG.\mathsf{read}()$. Informally, the $REG.\mathsf{write}()$ operation updates the value stored in the shared variable while the $REG.\mathsf{read}()$ 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, called the invocation time and the reply time, with respect to the fictional global time. In the following $t_B(op)$ and $t_E(op)$ will respectively denote the invocation and reply times of operation op (i.e., $op = REG.\mathsf{write}()$ or $op = REG.\mathsf{read}()$).

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 entity 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 the following. On the other hand, an operation that does not terminate is said *failed*.

4.1 Classical Distributed Registers

The semantic of a distributed shared register can be classified as *safe*, *regular* or *atomic* [14]. In this paper, we will refer mainly to the *safe* and *regular* semantics. 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. The *safe* distributed register is defined by the following properties:

- *Liveness*: Any invocation of $REG.\mathsf{write}()$ or $REG.\mathsf{read}()$ eventually terminates.
- *Safety*: A $REG.\mathsf{read}()$ operation returns the last value written before its invocation (*i.e.* the value written by the latest write preceding this read operation), or any value of the register domain in case the read() operation is concurrent to a write() operation.

The *regular* distributed register is defined by the following properties:

- *Liveness*: Any invocation of $REG.\mathsf{write}()$ or $REG.\mathsf{read}()$ eventually terminates.

– *Safety*: A REG.read() operation returns the last value written before its invocation (*i.e.* the value written by the latest write preceding this read operation), or a value written by a write operation concurrent with it.

An *atomic* register is a regular register that verifies 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.

4.2 Extension to Stabilising Distributed Registers

Recently, classical registers definitions [14] have been extended to the self-stabilising area [4] for which the system can be hit by arbitrary errors. We assume that there is a time τ_{1w} at which the first write operation invoked in the system terminated.
The *stabilising safe register* is defined by the following properties:

– *Liveness.* Any invocation of REG.write() or REG.read() terminates.
– *Eventual safety.* There is a finite time $\tau_{stab} > \tau_{1w}$ after which each REG.read() r returns a value v that was written by a REG.write() operation w such that (a) w is the last write operation executed before r, or (b) v is any value in the register domain if a write operation is concurrent with r.

The *stabilising regular register* is defined by the following properties:

– *Liveness.* Any invocation of REG.write() or REG.read() terminates.
– *Eventual regularity.* There is a finite time $\tau_{stab} > \tau_{1w}$ after which each REG.read() r returns a value v that was written by a REG.write() operation w such that (a) w is the last write operation executed before r, or (b) w is a write operation concurrent with r.

Similarly, the *stabilising atomic register* is the eventual version of the atomic register defined above.

4.3 Bitcoin and Distributed Shared Registers

Interestingly enough, none of these definitions capture the behaviour of the Bitcoin blockchain. 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 increases its size. Looking at the stabilising 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.

Therefore, we need to further extend the distributed shared registers specification to a new register, which captures the semantics of Bitcoin. We call this new register the *Distributed Ledger Register*. We first show that the Distributed Ledger Register satisfies the regular properties and then prove that the Bitcoin blockchain maintenance verifies the Distributed Ledger Register properties.

5 Distributed Ledger Register

In this section, we aim at specifying a new type of read/write register that mimics the behaviour of the Bitcoin distributed ledger (i.e., Bitcoin blockchain), and that must be both writable and readable by any number of miners. In the following, this new register will be named the multi-writer multi-reader *Distributed Ledger Register*, or simply the *DLR*. Prior to formalising the properties of the distributed ledger register, we first illustrate its functioning.

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 invokes a write operation with $\mathcal{B}$ as parameter. The miner broadcasts $\mathcal{B}$ in the system. 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.

Let us now formalise the operations and the properties guaranteed by the distributed ledger register. The 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 its longest sequence of blocks, starting from the root. The value of the DLR is called the blockchain and is denoted by $\mathcal{B}$. The DLR is equipped with a write and read operations. The DLR.write operation allows any miner to try to change the value of DLR with value $\mathcal{B}$, where $\mathcal{B}$ is a sequence of blocks. The DLR.read() operation allows any miner to retrieve the value of DLR.

Note 1. Note that the value returned by the read() operation is different in Bitcoin and Ethereum. In Bitcoin, the longest chain is returned while in Ethereum the heaviest one is returned.

In order to take into account the level of confirmation of a block (see Section 2), we introduce the notion of a k-valid write.

Definition 1 (k-valid write). *Operation DLR(k).write($\mathcal{B}$) is k-valid if and only if there exist a time $t > 0$ and an integer $k > 0$ such that a virtual DLR(k).read()*

*invoked at time $t' > t$ after the invocation of DLR(k).*write*($\mathcal{B}$) returns a chain $\mathcal{B}'$ such that $\mathcal{B}$ is a prefix of $\mathcal{B}'$ and* length$(\mathcal{B}') \geq$ length$(\mathcal{B}) + k$*, where function* length*($\mathcal{B}$) returns the number of blocks that compose chain $\mathcal{B}$.*

Operation DLR.write($\mathcal{B}$) returns *true* if DLR.write($\mathcal{B}$) 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 [17], 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.

5.1 Specification of the Distributed Ledger Register

A DLR multi-reader multi-writer register is defined by the following properties.

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

As recalled in the previous section, the semantic of a distributed shared register can be classified as *safe*, *regular* or *atomic* according to the returned values read in presence of concurrent writes [14]. In the following we establish the relation between those classical registers and the newly defined distributed ledger register.

Theorem 1. *The Distributed Ledger Register verifies the regular register semantic.*

Proof. The liveness property of DLR register being identical to the liveness property for the regular register, we only need to prove that the distributed ledger register verifies the safety property of the regular register.

Consider a read operation of DLR r that is not concurrent with any write operations. By the *k-coherency* property, the value $\mathcal{B}$ returned by r is a value whose prefix $\mathcal{B}'$ is the value of the register written by the last k-valid DLR.write($\mathcal{B}'$) operation that preceded r. Let w be this k-valid write operation. By construction r returns the value written by w, which makes the safety property of regularity satisfied. Now suppose that r is concurrent with write operations that started after operation w. By the *k-coherency* property, r may return any of the chains written by these writes. However, all these chains have as common prefix the chain written by w, which completes the proof.

Theorem 2. *The Distributed Ledger Register does not verify the atomic register semantic.*

Proof. From Theorem 1 DLR verifies the regular register specification. We now show that DLR does not verify the *no new/old inversion* property. Consider two read operations $r1$ and $r2$ such that $r1$ happens before $r2$. Let w=DLR.write($\mathcal{B}$) be the last k-valid write that precedes $r1$ and $r2$. Consider two different k-valid write operations $w1$=DLR.write($\mathcal{B}'$) and $w2$=DLR.write($\mathcal{B}''$) that happen after w and that are concurrent with $r1$ and $r2$. By definition of k-validity, $\mathcal{B}$ is a prefix for both $\mathcal{B}'$ and $\mathcal{B}''$, while both $\mathcal{B}'$ and $\mathcal{B}''$ are different. By the *k-coherency* property, $r1$ may return $\mathcal{B}'$ while $r2$ may return $\mathcal{B}''$ which violates the no new/old inversion property.

5.2 Bitcoin and the Distributed Ledger Register

The DLR-Algorithm below describes the maintenance of the Bitcoin blockchain in terms of read/write invocations over the blockchain tree. Each miner manages one local variable, called $\mathcal{TB}$, that stores the blockchain tree, and has access to two functions, the best_chain function whose argument is ($\mathcal{TB}$), and the update_tree() functions whose arguments are $\mathcal{TB}$ and a sequence of blocks $\mathcal{B}$. Specifically,

- Function best_chain($\mathcal{TB}$) returns the longest chain of $\mathcal{TB}$ starting from the genesis block.
- Function update_tree($\mathcal{TB}$, $\mathcal{B}$) fusions $\mathcal{TB}$ with the sequence of $\mathcal{B}$. Specifically, if $\mathcal{TB}$ contains a branch which prefixes $\mathcal{B}$, then this branch is replaced by $\mathcal{B}$, otherwise $\mathcal{B}$ is added to $\mathcal{TB}$. Note that $\mathcal{B}$ must be well-formed and must start with genesis block.

As described above, the DLR-algorithm, whose pseudo-code appears in Figure 2, run by a miner is quite simple. The block creation process requires that a miner invokes the DLR.read() on $\mathcal{TB}$ to get the best chain $\mathcal{B}$ (see Figure 3). From $\mathcal{B}$, the miner can create its block (by solving the required proof-of-work), appends it to $\mathcal{B}$, and invokes the DLR.write($\mathcal{B}$) on $\mathcal{TB}$ (see Figure 3). This operation updates its local tree, and then diffuse the updated longest chain in the network by invoking the broadcast primitive. The DLR.write($\mathcal{B}$) operation does not return until the new potential block is valid, i.e. k other blocks have been appended to the local tree after it. Therefore, the miner will read its local tree until the above condition is verified. The DLR-algorithm assumes that miners continuously DLR.write new blocs otherwise the liveness of the algorithm would not hold, as shown in the sequel.

It may happen that, due to concurrent writes, the longest returned blockchain has not $\mathcal{B}$ as a prefix. In that case the miner knows that its DLR.write($\mathcal{B}$) operation is not successful, i.e., returns abort. It returns true otherwise.

We now prove that DLR-Algorithm conditionally verifies the distributed ledger register properties.

Lemma 1. *DLR-Algorithm verifies the liveness property of the DLR register.*

```
DLR-Algorithm  % run by a miner %

(01)  $\mathcal{B}$ = DLR.read()
(02)  create the well-formed block $b$ from $\mathcal{B}$
(03)  append $b$ to $\mathcal{B}$
(04)  DLR.write($\mathcal{B}$)
(05)  return
```

Fig. 2. Algorithm run by any miner

```
Operation DLR.read () is % issued by a reader %
(01)  return(best_chain($\mathcal{TB}$) )

Operation DLR.write ($\mathcal{B}$) is % issued by a writer %
(02)  update_tree($\mathcal{TB}$, $\mathcal{B}$)
(03)  broadcast (<propose $\mathcal{B}$>)
(04)  repeat
(05)      $\mathcal{B}'$ = DLR.read ()
(06)  until length($\mathcal{B}'$) $\geq$ length($\mathcal{B}$) + $k$
(07)  if  $\mathcal{B}$= prefix($\mathcal{B}'$) return true
(08)  else return abort
----------------------------------------------------------
(09)  upon deliver(<propose $\mathcal{B}$>) update_tree($\mathcal{TB}$, $\mathcal{B}$)
```

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

Proof. The liveness property is trivial and follows directly from the code. Indeed, a DLR.read() operation always returns since the read is executed locally. For the write operations the only blocking part of the code is the repeat loop. By assumption of the DLR-Algorithm, miners continuously try to create blocks which gives rise to the invocation of the DLR.write() operation every 10 in expectation. Thus the loop stops, which allows the DLR.write() operation to either return true or abort, which terminates the DLR-Algorithm.

Lemma 2. *Each non aborted* DLR.write() *invoked by the DLR-Algorithm verifies the k-validity property.*

Proof. Let w be any non-aborting DLR.write() operation that writes some chain $\mathcal{B}$ at time say $t > 0$. Note that this operation returns only when the best chain in the $\mathcal{TB}$ tree, say $\mathcal{B}'$, has $\mathcal{B}$ as a prefix and has at least k additional blocks. Let r be a DLR.read() that happens after w. If r is invoked by the same miner than the property trivially follows. Assume now that r has been invoked by a miner different from the writer. By the τ-delivery property of the broadcast primitive, there is a time $t' > \tau + t$ such that $\mathcal{B}'$ has reached every miner in the system. Hence any read r invoked after $\tau + t$ returns $\mathcal{B}'$.

Lemma 3. *DLR-Algorithm verifies the k-coherency property of the DLR register under the hypothesis that each read is invoked after that τ time units have elapsed since the last k-valid write.*

Proof. Let r be a read() operation invoked at time t'. Let w be the last k-valid write that happened before r at time $t < t'$. At t, the longest chain read by w is $\mathcal{B}'$. By the the τ-delivery property of the boradcast primitive, then in the worst case at time $t + \tau$, chain $\mathcal{B}$ reaches every miner in the system, and in particular the reader. Any read() invoked at $t' \geq t + \tau$ verifies the k-coherency property. Note that a read() operation invoked at $t \leq t' < t + \tau$ may return the last k-valid write that happened before w .

The following theorem is a direct consequence of the three above lemmata.

Theorem 3. *DLR-Algorithm verifies the DLR specification under the hypothesis that each* read *is invoked after that τ time units have elapsed since the last k-valid* write.

Note that when reads are invoked without any constraints the DLR-Algorithm does not verify the k-coherency.

6 Conclusions and Open Questions

In this paper we have shown that classical distributed shared registers do not capture totally the behaviour of Bitcoin ledger, which has led us to propose a specification of a distributed ledger register with a regular flavour.

We have then proven that the blockchain maintenance of Bitcoin verifies the distributed ledger register specification under strict conditions and only in partially synchronous systems. The first conclusion of our study is that Bitcoin does not implement strong coherency criteria even in partially synchronous systems. This finding explains the constant adjustments that Bitcoin experienced since its creation.

Our paper opens several research directions. The implementation of the distributed ledger register with strong coherency guarantees (i.e. similar to the *linearisability*) in a adversarial asynchronous environment is a real challenge that might be mitigated by relying on tools such as k-quorums abstraction defined in [2]. Another interesting research direction is the identification of the minimal building blocks necessary to implement a blockchain-based transactional system in an adversarial model.

Acknowledgements

The authors would like to thank Sara Tucci Piergiovanni and Antonella del Pozzo for insightful comments on a preliminary version of this paper.

References

1. Ethereum Stack Exchange. https://ethereum.stackexchange.com/questions/319/what-number-of-confirmations-is-considered-secure-in-ethereum, 2016.
2. A. S. Aiyer, L. Alvisi, and R. A. Bazzi. Byzantine and multi-writer k-quorums. In *Distributed Computing, 20th International Symposium, DISC 2006, Stockholm, Sweden, September 18-20, 2006, Proceedings*, pages 443–458, 2006.
3. E. Anceaume, T. Lajoie-Mazenc, R. Ludinard, and B. Sericola. Safety Analysis of Bitcoin Improvement Proposals. In *15th IEEE International Symposium on Network Computing and Applications (NCA)*, 2016.
4. S. Bonomi, S. Dolev, M. Potop-Butucaru, and M. Raynal. Stabilizing server-based storage in byzantine asynchronous message-passing systems: Extended abstract. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*, pages 471–479, 2015.
5. C. Cachin. Blockchain - From the Anarchy of Cryptocurrencies to the Enterprise (Keynote Abstract). In *Proc. of the OPODIS International Conference*, 2016.
6. K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. E. Kosba, A. Miller, P. Saxena, E. Shi, E. Gün Sirer, D. Song, and R. Wattenhofer. On scaling decentralized blockchains - (A position paper). In *Financial Cryptography and Data Security - FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Christ Church, Barbados, February 26, 2016, Revised Selected Papers*, pages 106–125, 2016.
7. C. Decker, J. Seidel, and R. Wattenhofer. Bitcoin Meets Strong Consistency. In *Proc. of the ICDCN International Conference*, 2016.
8. C. Decker, J. Seidel, and R. Wattenhofer. Bitcoin Meets Strong Consistency. In *17th International Conference on Distributed Computing and Networking (ICDCN)*, 2016.
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. M. Herlihy and M. Moir. Blockchains and the logic of accountability: Keynote address. In *Proc. of the ACM/IEEE LICS Symposium*, 2016.
12. 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.
13. 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.
14. L. Lamport. On inter-process communications, part I: basic formalism and part II: algorithms. *Distributed Computing*, 1(2):77–101, 1986.
15. A. Miller and J. J. LaViola Jr. Anonymous byzantine consensus from moderately-hard puzzles: A model for bitcoin. http://bravenewcoin.com/assets/Whitepapers/, 2014.
16. S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/bitcoin.pdf, 2008.
17. S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.
18. Pass R., Seeman L., and Shelat A. Analysis of the blockchain protocol in asynchronous networks. In *Proc. of the EUROCRYPT International Conference*, 2017.

19. G. Wood. Ethereum: A secure decentralised generalised transaction ledger. http://gavwood.com/Paper.pdf.