Blockchain abstract data type
Emmanuelle Anceaume, Antonella del Pozzo, Romaric Ludinard, Maria Potop-Butucaru, Sara Tucci-Piergiovanni

To cite this version:

HAL Id: hal-02380364
https://cnrs.hal.science/hal-02380364
Submitted on 26 Nov 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.
Blockchain Abstract Data Type

Emmanuelle Anceaume†, Antonella Del Pozzo*, Romaric Ludinard**, Maria Potop-Butucaru†, Sara Tucci-Piergiovanni†

†CNRS, IRISA
*CEA LIST, PC 174, Gif-sur-Yvette, 91191, France
** IMT Atlantique, IRISA
†Sorbonne Université, CNRS, Laboratoire d’Informatique de Paris 6, LIP6, Paris, France

Abstract—The presented work continues the line of recent distributed computing community efforts dedicated to the theoretical aspects of blockchains. This paper is the first to specify blockchains as a composition of abstract data types all together with a hierarchy of consistency criteria that formally characterizes the histories admissible for distributed programs that use them. Our work is based on an original oracle-based construction that, along with new consistency definitions, captures the eventual convergence process in blockchain systems. The paper presents as well some results on implementability of the presented abstractions and a mapping of representative existing blockchains from both academia and industry in our framework.

I. INTRODUCTION

The paper proposes a new data type to formally model blockchains and their behavior. We aim at providing 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 [5], [4]. The question is about the consistency criterion of blockchains as Bitcoin [19] and Ethereum [24] that technically do not solve Consensus, and their relation with Consensus in general.

We advocate that the key point to capture blockchain behaviors is to define consistency criteria allowing mutable operations to create forks and restricting the values read, i.e. modeling the data structure as an append-only tree and not as an append-only queue. This way we can easily define semantics equivalent to eventually consistent append-only queue but as well as weaker semantics. More in detail, we define a semantic equivalent to eventually consistent append-only queue but by restricting any two reads to return two chains such that one is the prefix of the other. We call this consistency property Strong Prefix (already introduced in [14]). Additionally, we define a weaker semantics restricting any two reads to return chains that have a divergent prefix for a finite interval of the history. We call this consistency property Eventual Prefix. Note that our consistency criteria specifically defined for blockchain systems have a similarity flavor with fork-consistency defined in [18], which concern a different area, i.e., the data integrity in the network file system domain.

Another peculiarity of blockchains lies in the notion of validity of blocks, i.e. the blockchain must contain only blocks that satisfy a given predicate. Let us note that validity can be achieved through proof-of-work (Dwork and Naor [11]) or other agreement mechanisms. We advocate that to abstract away implementation-specific validation mechanisms, the validation process must be encapsulated in an oracle model separated from the process of updating the data structure. Because the oracle is the only generator of valid blocks and only valid blocks can be appended to the tree, it follows that, it is the oracle that grants the access to the data structure and it might also own a synchronization power to control the size of forks, i.e., the number of blocks that point back to the same block of the tree. In this respect we define oracle models such that, depending on the model, the size $k$ of forks can be equal to 1 (i.e., strongest oracle model), strictly greater than 1, or unbounded (i.e., weakest oracle model).

The blockchain is thus abstracted by an oracle-based construction in which the update and consistency of the tree data structure depends on the validation and synchronization power of the oracle.

The main contribution of the paper is a formal unified framework providing blockchain consistency criteria that can be combined with oracle models in a proper hierarchy of abstract data types [23] independent of the underlying communication and failure models. Thanks to the establishment of the formal framework the following implementability results are shown.

- The strongest oracle, guaranteeing no fork, has Consensus number $\infty$ in the Consensus hierarchy of concurrent objects [15] (Theorem V.2). Note that similarly to [8], [13], [7] we extend the validity property of Consensus to fit the blockchain setting.
- The weakest oracle, which validates a potentially unbounded number of blocks to be appended to a given block, is not stronger than Generalized Agreement Lattice [12].
- The impossibility to guarantee Strong Prefix in a message-passing system if forks of size $k > 1$ are allowed (Theorem V.6). This means that Strong Prefix needs the strongest oracle to be implemented, which is at least as strong as Consensus.
- A necessary condition (Theorem V.5) for Eventual Prefix in a message-passing system is that each update sent by a correct process must be eventually received by every correct process. Moreover, the result implies that it is impossible to implement Eventual Prefix if even a single
update is dropped at some correct process while it has been received at all the other correct processes.

The proposed framework along with the above-mentioned results helps in classifying existing blockchains in terms of their consistency and implementability. We used the framework to classify several blockchain proposals. We showed that Bitcoin [19] and Ethereum [24] have a validation mechanism that maps to our weakest oracle and then they only implement eventual prefix, while other proposals map to our strongest oracle, falling in the class of those that guarantee Strong Prefix (e.g. Hyperledger Fabric [4], PeerCensus [9], ByzCoin [16], see Section V-C for further details).

Note that for space reasons all the theorems and lemmas proofs and some formal definitions do not appear in this article but are presented in the supplementary materials [3].

II. RELATED WORK

In [20] the authors extract Bitcoin backbone and define invariants 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 [21]. 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 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, [5] proposes a formalization of distributed ledgers modeled as an ordered list of records. The authors propose in their formalization 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. On the other hand, the proposed formalization does not propose weaker consistency semantics more suitable for proof-of-work blockchains as BitCoin. The work achieved in [5] is complementary to the one presented in [2], where the authors study the consistency of blockchain by modeling it as a register. Finally, [14] presents 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.

III. PRELIMINARIES ON SHARED OBJECT SPECIFICATIONS BASED ON ABSTRACT DATA TYPES

The basic idea underlying the use of abstract data types is to specify shared objects using two complementary facets [22]: a sequential specification that describes the semantics of the object, and a consistency criterion over concurrent histories, i.e. the set of admissible executions in a concurrent environment. A. Abstract Data Type (ADT)

The model used to specify an abstract data type is a form of transducer, as Mealy’s machines, accepting an infinite but countable number of states. In the following, an abstract data type refers to a 6-tuple \( T = (A, B, Z, \xi_0, \tau, \delta) \). The values that can be taken by the data type are encoded in the abstract state, taken from a set \( Z \). We refer by \( \xi_0 \in Z \) the initial state of the ADT. It is possible to access the object using the symbols of an input alphabet \( A \). Unlike the methods of a class, the input symbols of the abstract data type do not have arguments. Indeed, as one authorizes a potentially infinite set of operations, the call of the same operation with different arguments is encoded by different symbols. An operation can have two types of effects. First, it can have a side-effect that changes the abstract state according to the transition system formalized by a transition function \( \tau \). Second, operations can return values taken from an output alphabet \( B \), which depends on the state in which they are called and an output function \( \delta \).

For example, the pop operation in a stack removes the element at the top of the stack and returns that element (its output).

B. Sequential specification of an ADT

An abstract data type, by its transition system, defines the sequential specification of an object. The sequential specification of an object describes its behavior when its operations are applied sequentially. That is, if we consider a path that traverses its system of transitions, then the word formed by the subsequent labels on the path is part of the sequential specification of the abstract data type, i.e. it is a sequential history. A sequential history of an ADT \( T \) refers to a sequence \( (\sigma_i)_{i \geq 0} \) (finite or not) of operations leading the state of \( T \) to evolve according to its specification [3].

1) Concurrent histories of an ADT: Concurrent histories are defined considering a partial order relation among events executed by different processes. A set of processes invoking operations of an ADT defines a concurrent history. Operations are not executed instantaneously, i.e., given an operation \( o \in \Sigma = A \cup (A \times B) \), we denote by \( e_{inv}(o) \) the invocation event of operation \( o \) and by \( e_{rsp}(o) \) the corresponding response event. In addition, we denote by \( e_{rsp}(o) : x \) the returned value associated to the response event \( e_{rsp}(o) \). In the following \( E \) represents the set of events and \( \Lambda \) is the function which associates events to the operations in \( \Sigma \). Given two events \( (e, e') \in E^2 \) we say that \( e \rightarrow e' \) in the process order if they are produced by the same process, \( e \neq e' \) and \( e \) happens before \( e' \). Given two events \( e, e' \in E \), we say that \( e \) precedes \( e' \) in operation order, denoted by \( e \prec e' \), if \( e' \) is the invocation of an operation occurred at time \( t' \) and \( e \) is the response of another operation occurred at time \( t \) with \( t < t' \). Finally, for any couple of events \( (e, e') \in E^2 \) with \( e \neq e' \), we say that \( e \) precedes \( e' \) in program order, denoted by \( e \rightarrow' e' \), if \( e \rightarrow e' \) or \( e \prec e' \). These asymmetric event structures allow us to define a concurrent history of an ADT \( T = (A, B, Z, \xi_0, \tau, \delta) \) as a 6-tuple \( H = (\Sigma, E, \Lambda, \rightarrow, \prec, \rightarrow') \) [3].

2) Consistency criterion: A consistency criterion characterizes which concurrent histories are admissible for a given
abstract data type. It can be viewed as a function that associates a concurrent specification to abstract data types. Given a consistency criterion $C$, an algorithm $A_T$ implementing the ADT $T$ is $C$-consistent if all the operations terminate and all the admissible executions are $C$-consistent, i.e. they satisfy consistency criterion $C$.

IV. BlockTree and Token Oracle ADTs

In this section we present the BlockTree and the token Oracle ADTs along with consistency criteria.

A. BlockTree ADT

We formalize the data structure implemented by blockchain-like systems as a directed rooted tree $bt = (V_{bt}, E_{bt})$ called BlockTree. Each vertex of the BlockTree is a block and any edge points backward to the root, called genesis block. By convention, the root of the BlockTree is denoted by $b_0$. Two operations are provided: the append($b_{by}$) operation, which appends a new block $b_{by}$ to the BlockTree, and the read($()$) operation, which returns a sequence of blocks of the BlockTree. This sequence of blocks is called the blockchain and is selected according to function $f$ (see below). Only blocks satisfying some validity predicate $P$ can be appended to the BlockTree. Predicate $P$ is application dependent. Predicate $P$ mainly abstracts the creation process of a block, which may fail (return false). Note that false is denoted by $\bot$ or successfully terminate (returns true, denoted by $\top$). For instance, in Bitcoin, a block is considered valid if it can be connected to the current blockchain and does not contain double-spending transactions.

We represent by $B$ a countable and non empty set of blocks and by $B' \subseteq B$ a countable and non empty set of valid blocks, i.e., $\forall b \in B', P(b) = \top$. By assumption $b_0 \in B'$; we also denote by $BC$ a countable non empty set of blockchains, where a blockchain is a path from a leaf of $b_0$ to $b_0$. A blockchain is denoted by $bc$. Finally, $F$ is a countable non empty set of selection functions, $f \in F : BT \rightarrow BC$; $f(b)$ selects a sequence of blocks $bc$ from the BlockTree $bt$ (note that $b_0$ is not returned) and if $bt = b_0$ then $f(b_0) = b_0$. This reflects for instance the longest chain or the heaviest chain used in some blockchain implementations. The selection function $f$ and the predicate $P$ are parameters of the ADT which are encoded in the state and do not change over the computation. The following notations are used: $\{b_0\} \sim f(b)$ represents the concatenation of $b_0$ with the blockchain of $bt$; and $\{b_0\} \sim f(b) \sim \{b\}$ represents the concatenation of $b_0$ with the blockchain of $bt$ and a block $b$.

1) Sequential specification of the BlockTree: The sequential specification of the BlockTree is defined as follows.

Definition IV.1 (BlockTree ADT (BT-ADT)). The BlockTree Abstract Data Type is the 6-tuple BT-ADT=$\{A = \{\text{append}(b_{by}, b_{by}), \text{read}(): b \in B\}, B = BC \cup \{\text{true}, \text{false}\}, Z = BT \times F \times (B \rightarrow \{\text{true}, \text{false}\}), \xi_0 = (b_0, f, P), \tau, \delta\}$, where the transition function $\tau : Z \times A \rightarrow Z$ is defined by

$$\tau((bt, f, P), \text{read}()) = (bt, f, P)$$

$$\tau((bt, f, P), \text{append}(b)) = \begin{cases} \{(b_0) \sim f(bt) \sim \{b\}, f, P\} & \text{if } b \in B' \\ (bt, f, P) & \text{otherwise} \end{cases}$$

and the output function $\delta : Z \times A \rightarrow B$ is defined by

$$\delta((bt, f, P), \text{read}()) = \begin{cases} \{b_0\} & \text{if } bt = b_0 \\ \{b_0\} \sim f(bt) & \text{otherwise} \end{cases}$$

$$\delta((bt, f, P), \text{append}(b)) = \begin{cases} \text{true} & \text{if } b \in B' \\ \text{false} & \text{otherwise} \end{cases}$$

The semantic of the read and the append operations directly depends on the selection function $f \in F$. In this work we let this function generic to suit the different blockchain implementations. Figure 1 illustrates an execution of the BT-ADT $bt$. Starting from the initial state $\xi_0$, state $\xi_1$ is obtained by appending block $b_1$ to $\xi_0$ and state $\xi_2$ is obtained by appending block $b_2$ to $\xi_1$.

The read operation applied in state $\xi_1$ returns blockchain $\{b_0\} \sim \{b_1\}$, and the read applied in state $\xi_2$ returns blockchain $\{b_0\} \sim f(b) \sim \{b_2\} = \{b_0\} \sim \{b_1\} \sim \{b_2\}$.

2) Concurrent histories of a BT-ADT and consistency criteria: A BT-ADT consistency criterion is a function that returns the set of concurrent histories admissible for a BlockTree abstract data type. We define two BT consistency criteria: BT Strong consistency and BT Eventual consistency. For ease of readability, we employ the following notations:

- $E(a^*, r^*)$ refers to an infinite set containing an infinite number of append() and read() invocation and response events. Similarly, $E(a, r^*)$ refers to an infinite set containing (i) a finite number of append() invocation and response events and (ii) an infinite number of read() invocation and response events;
- $\text{score} : BC \rightarrow \mathbb{N}$ denotes a monotonically increasing deterministic function that takes as input a blockchain $bc$ and returns a natural number $s$ as score of $bc$, which can be the depth, the weight, etc of $bc$. Informally we refer to such value as the score of a blockchain; by convention we refer to the score of the blockchain uniquely composed by the genesis block as $s_0$, i.e. $\text{score}(\{b_0\}) = s_0$. Increasing monotonicity means that $\text{score}(bc \sim \{b\}) > \text{score}(bc)$;
- $\text{mcps} : BC \times BC \rightarrow \mathbb{N}$ is a function which, given two blockchains $bc$ and $bc'$ returns the score of the maximal common prefix of $bc$ and $bc'$;
- $bc \sqsubseteq bc'$ iff $bc$ prefixes $bc'$.

We now present the BT Strong Consistency criterion. Informally it says that any two read() operations return blockchains such that one is the prefix of the other. This is formalized through the following four properties.

The Block validity property imposes that each block in a blockchain returned by a read() operation is valid (i.e.,
satisfies predicate \( P \) and has previously been inserted in the BlockTree with the \( \text{append}() \) operation. Formally,

**Definition IV.2** (Block validity). \( \forall e_{\text{rsp}}(r) \in E, \forall b \in e_{\text{rsp}}(r) : bc, b \in B' \land \exists e_{\text{inv}}(\text{append}(b)) \in E, e_{\text{inv}}(\text{append}(b)) \not\succ e_{\text{rsp}}(r) \)

The Local monotonic read property states that, given the sequence of \( \text{read}() \) operations at the same process, the score of the returned blockchain never decreases; Formally,

**Definition IV.3** (Local monotonic read). \( \forall e_{\text{rsp}}(r), e_{\text{rsp}}(r') \in E^2, \text{if } e_{\text{rsp}}(r) \rightarrow e_{\text{rsp}}(r'), \text{then } \text{score}(e_{\text{rsp}}(r) : bc) \leq \text{score}(e_{\text{rsp}}(r') : bc') \)

The Strong prefix property says that for each pair of read operations, one of the returned blockchains is a prefix of the other returned one. Formally,

**Definition IV.4** (Strong prefix). \( \forall e_{\text{rsp}}(r), e_{\text{rsp}}(r') \in E^2, (e_{\text{rsp}}(r') : bc') \subseteq e_{\text{rsp}}(r) : bc ) \lor (e_{\text{rsp}}(r) : bc \subseteq e_{\text{rsp}}(r') : bc') \)

Finally, the Ever growing tree states that scores of returned blockchains eventually grow. More precisely, let \( s \) be the score of the blockchain returned by a read response event \( r \) in \( E(a^*, r^*) \), then for each \( \text{read}(r) \) operation \( r \), the set of \( \text{read}(r) \) operations \( r' \) such that \( e_{\text{rsp}}(r) \not\succ e_{\text{inv}}(r') \) that do not return blockchains with a score greater than \( s \) is finite. Formally,

**Definition IV.5** (Ever growing tree). \( \forall e_{\text{rsp}}(r) \in E(a^*, r^*), s = \text{score}(e_{\text{rsp}}(r) : bc) \text{ then } |\{e_{\text{inv}}(r') \in E \mid e_{\text{inv}}(r') \not\succ e_{\text{inv}}(r'), \text{score}(e_{\text{rsp}}(r') : bc') \leq s\}| < \infty \)

**Definition IV.6** (BT Strong Consistency (SC) criterion). A concurrent history \( H = (\Sigma, E, \Lambda, \rightarrow, \prec, \not\prec) \) of the system that uses a BT-ADT verifies the BT Strong Consistency criterion if the Block validity, Local monotonic read, Strong prefix and the Ever growing tree properties hold.

We now present the BT Eventual Consistency criterion, a weaker version of the Strong Consistency criterion. Informally, the BT Eventual Consistency criterion says that eventually any two \( \text{read}(r) \) operations return blockchains that share the same prefix, which differs from the BT Strong Consistency criterion by the Eventual prefix property. The Eventual prefix property says that for each blockchain returned by a \( \text{read}(r) \) operation with \( s \) as score, then eventually all the \( \text{read}(r) \) operations will return blockchains sharing the same maximum common prefix at least up to \( s \). Say differently, let \( H \) be a history with an infinite number of \( \text{read}(r) \) operations, and let \( s \) be the score of the blockchain returned by a \( \text{read}(r) \) operation \( r \) then, the set of \( \text{read}(r) \) operations \( r' \), such that \( e_{\text{rsp}}(r) \not\succ e_{\text{inv}}(r') \), that do not return blockchains sharing the same prefix at least up to \( s \) is finite. We formalise this notion as follows:

**Definition IV.7** (Eventual prefix property). Given a concurrent history \( H = (\Sigma, E(a, r^*), \Lambda, \rightarrow, \prec, \not\prec) \) of the system that uses a BT-ADT, we denote by \( s \), for any \( \text{read}(r) \) operation \( r \in \Sigma \) such that \( \exists e \in E(a, r^*), \Lambda(r) = e \), the score of the returned blockchain, i.e., \( s = \text{score}(e_{\text{rsp}}(r) : bc) \). We denote by \( E_r \) the set of response events of \( \text{read}(r) \) operations that occurred after \( r \) response, i.e. \( E_r = \{ e \in E \mid \exists r' \in \Sigma, r' = \text{read}, e = e_{\text{rsp}}(r') \land e_{\text{inv}}(r') \not\succ e_{\text{rsp}}(r') \} \). Then, \( H \) satisfies the Eventual prefix property if for all \( \text{read}(r) \) operations \( r \in \Sigma \) with score \( s \), there is a set \( S = \{ (e_{\text{rsp}}(r_h), e_{\text{rsp}}(r_k)) \in E^2 \mid h \neq k, \text{mcps}(e_{\text{rsp}}(r_h) : bc_h, e_{\text{rsp}}(r_k) : bc_k) < s \} \) and \( |S| < \infty \).

**Definition IV.8** (BT Eventual Consistency (EC) criterion). A concurrent history \( H = (\Sigma, E, \Lambda, \rightarrow, \prec, \not\prec) \) of the system that uses a BT-ADT verifies the BT Eventual Consistency criterion if it satisfies the Block validity, Local monotonic read, Ever growing tree, and the Eventual prefix properties.

3) Relationships between Eventual Consistency and Strong Consistency: Let \( \mathcal{H}_{\text{EC}} \) and \( \mathcal{H}_{\text{SC}} \) be the set of histories satisfying respectively the EC and the SC consistency criteria.

**Theorem IV.1.** Any history \( H \) satisfying SC criterion satisfies EC and \( \exists H \) satisfying EC that does not satisfy SC, i.e., \( \mathcal{H}_{\text{SC}} \subset \mathcal{H}_{\text{EC}} \).

Proof of Theorem IV.1 and an illustration showing a BT Eventually Consistent history which is not Strongly Consistent are reported in the supplementary materials [3].

Let us remark that the BlockTree allows at any time to create a new branch in the tree, which is called a fork in the blockchain literature. Note that that histories with no append operations are trivially admitted.

In the following we introduce a new abstract data type called Token Oracle, which when combined with
the BlockTree will help in (i) validating blocks and (ii) controlling the presence of forks and their number, if any.

B. Token oracle \( \Theta \)

We now formalize the Token Oracle \( \Theta \) to capture the creation of blocks in the BlockTree structure. The block creation process requires that each new block must be closely related to an already existing valid block in the BlockTree structure. We abstract this implementation-dependent process by assuming that a process will obtain the right to chain a new block by to \( b_h \in B' \), if it successfully gains a token \( tkn_h \) from the token oracle \( \Theta \). Once obtained, the proposed block \( b_i \) is considered as valid, and will be denoted by \( bt_i^{tkn_h} \). By construction \( bt_i^{tkn_h} \in B' \). In the following, in order to be as much general as possible, we model blocks as objects. More formally, when a process wants to access some valid object \( obj_j \) such that \( P(obj_j) = T \) it invokes the getToken\((obj_j, obj_j)\) operation with object \( obj_j \) from set \( O = \{obj_1, obj_2, \ldots \} \). If getToken\((obj_j, obj_j)\) operation is successful, it returns the valid object \( obj_j^{tkn_h} \) such that \( tkn_h \) is the token required to validate valid object \( obj_j \). The set of valid objects is denoted by \( O' \), i.e., \( \forall obj_q \in O', P(obj_q) = T \). We say that a valid object is generated each time it is successfully returned by a getToken\((obj_j, obj_j)\) operation and it is consumed when the oracle grants the right to associate this valid object \( obj_j^{tkn_h} \) to \( obj_j \). In the following, once an object is valid, it is clear from the context, we will not explicitly the token \( tkn \) with makes the object valid.

A valid object \( obj_j^{tkn_h} \) is consumed through the consumeToken\((obj_j^{tkn_h})\) operation. No more than \( k \) valid objects \( obj_1^{tkn_h}, \ldots, obj_j^{tkn_h}, 1 \leq j \leq k \), can be consumed for \( obj_j \) where \( k \) is a parameter of the token oracle. The side-effect of the consumeToken\((obj_j^{tkn_h})\) on the state of the token oracle is the insertion of the valid object \( obj_j^{tkn_h} \) in a set related to \( obj_j \) as long as the cardinality of such set is less than or equal to \( k \).

We specify two token oracles, which differ in the way tokens are managed. The first oracle, called prodigal and denoted by \( \Theta_P \), has no upper bound on the number of tokens consumed for an object, while the second oracle \( \Theta_F \), called frugal, and denoted by \( \Theta_F \), guarantees that no more than \( k \) token can be consumed for each object.

The prodigal oracle \( \Theta_P \) when combined with the BlockTree abstract data type will only help in validating blocks, while the frugal oracle \( \Theta_F \) manages tokens in a more controlled way to guarantee that no more than \( k \) forks can occur on a given block.

For both oracles, when getToken\((obj_j, obj_j)\) operation is invoked, the oracle provides a valid object with a certain probability \( p_{\alpha_i} > 0 \) where \( \alpha_i \) is a “merit” parameter characterizing the invoking process \( i \). \(^1\) Note that the oracle knows \( \alpha_i \) of the invoking process \( i \), which might be unknown to the process itself. For each merit \( \alpha_i \), the state of the token oracle embeds an infinite tape where each cell of the tape contains either \( tkn \) or \( \bot \). Since each tape is identified by a specific \( \alpha_i \) and \( p_{\alpha_i} \), we assume that each tape contains a pseudorandom sequence of values in \( \{tkn, \bot\} \) depending on \( \alpha_i \). \(^2\)

When a getToken\((obj_j, obj_j)\) operation is invoked by a process with merit \( \alpha_i \), the oracle pops the first cell from the tape associated to \( \alpha_i \) and a valid object is provided to the process if that cell contains \( tkn \). Both oracles enjoy an infinite array of sets, one set for each valid object \( obj_j \), which is populated each time a valid object \( obj_j \) is consumed. When the set cardinality reaches \( k \) then no more tokens can be consumed for that object. For the sake of generality, \( \Theta_P \) is defined as \( \Theta_P \) with \( k = \infty \) while for \( \Theta_F \) a predetermined \( k \in \mathbb{N} \) is specified. Hence, the state of the token oracle contains \( (i) \) the infinite array \( K \) of sets (one per valid object) of elements in \( O' \), \( (ii) \) infinite tapes one for each possible merit, and \( (iii) \) the branching parameter \( k \). We consider oracles that are linearizable (with respect to their sequential specification): they behave as if all operations, including concurrent ones, are applied sequentially, so that each operation appears to take effect instantaneously as some point between their invocation and their response. The formal specification of \( \Theta_P \) and \( \Theta_F,k \) abstract data types can be found in the supplementary materials.

In Figure 2 is depicted a possible path of the transition system defined by \( \Theta_F,k \)-ADT and \( \Theta_P \)-ADT. When a process with merit \( \alpha_1 \) invokes getToken\((b_1, b_k)\), with \( b_1 \) the leaf of \( f(bt) \), the first cell of \( tape_{\alpha_1} \) is popped, if it contains a token, then getToken\((b_1, b_k)\) returns a valid block \( b_1^{tkn_1} \). Afterwards, when consumeToken\((b_1^{tkn_1})\) is invoked, the oracle checks if the cardinality of the set in \( K[1] \) is strictly smaller than \( k \), and if the affirmative inserts \( b_1^{tkn_1} \) in \( K[1] \). In any cases, consumeToken\(() \) returns the content of \( K[1] \), in this case \( b_1^{tkn_1} \). It follows that a process that gets a valid block for some block \( b_k \) but is not allowed to consume it, is anyway notified with the set of valid blocks that saturated \( K[h] \).

C. BT-ADT augmented with \( \Theta \) Oracles

We augment the BT-ADT with \( \Theta \) oracles and we analyze the histories generated by their combination. Specifically, we define a refinement of the append\((b_i)\) operation of the BT-ADT with the oracle operations as follows: the append\((b_i)\) operation triggers the getToken\((b_i) \leftarrow \text{last block}(f(bt)), b_i \) operation as long as it returns a valid block \( b_i^{tkn_n} \), and once obtained, the valid block might be consumed, and in any cases the append\((b_i)\) operation terminates. If less than \( k \) valid

\(^1\)The merit parameter can reflect for instance the hashing power of the invoking process.

\(^2\)We assume a pseudorandom sequence mostly indistinguishable from a Bernoulli sequence consisting of a finite or infinite number of independent random variables \( X_1, X_2, X_3, \ldots \) such that (i) for each \( k \), the value of \( X_k \) is either \( tkn \) or \( \bot \); and (ii) \( \forall X_k \), the probability that \( X_k = tkn \) is \( p_{\alpha_i} \).
blocks have already been consumed for \( b_0 \), the valid block is consumed i.e. block \( b_k^{k_{kn}} \) is appended to the block \( h \) in the blockchain \( f(h) \) (i.e., \( \{b_0\} \cap f(h) \cap \{b_1\} \)) and the append \((b_l)\) operation returns true, otherwise false. We say that the BT-ADT augmented with \( \Theta_P \) or \( \Theta_P \) oracle is a refinement \( \mathcal{R}(BT-ADT, \Theta_P) \) or \( \mathcal{R}(BT-ADT, \Theta_P) \) respectively. The formal specification of these refinements is given in the supplementary materials.

**Definition IV.9** \((k\text{-Fork coherence})\). A concurrent history 
\[ H = \langle \Sigma, E, \Lambda, \to, \prec, \succ \rangle \]
of \( \mathcal{R}(BT-ADT, \Theta_F,k) \) satisfies the \( k\text{-Fork coherence} \) if there are at most \( k \) append \((b_l^{k_{kn}})\) operations that return true for the same block \( b_l \).

**Theorem IV.2** \((k\text{-Fork Coherence})\). Any concurrent history 
\[ H = \langle \Sigma, E, \Lambda, \to, \prec, \succ \rangle \]
of \( \mathcal{R}(BT-ADT, \Theta_F,k) \) satisfies the \( k\text{-Fork Coherence} \).

### D. Hierarchy

We propose a hierarchy between BT-ADTs augmented with token oracle ADTs. We use the following notation: BT-ADT\(_{SC}\) and BT-ADT\(_{EC}\) to refer respectively to BT-ADT generating concurrent histories that satisfy the SC and the EC consistency criteria. When augmented with token oracles we get the following four typologies, where for the frugal oracle we explicit the value of \( k \): \( \mathcal{R}(BT-ADT_{SC}, \Theta_F,k) \), \( \mathcal{R}(BT-ADT_{SC}, \Theta_P) \), \( \mathcal{R}(BT-ADT_{EC}, \Theta_P) \), \( \mathcal{R}(BT-ADT_{EC}, \Theta_F,k) \). We aim at studying the relationships among the different refinements. Let \( \mathcal{R}^0(BT-ADT, \Theta_F,k) \) be the set of concurrent histories generated by a BT-ADT enriched with \( \Theta_F,k\)-ADT and \( \mathcal{R}(BT-ADT, \Theta_P) \) be the set of concurrent histories generated by a BT-ADT enriched with \( \Theta_P\)-ADT. Without loss of generality, we consider only the set of histories from which have been purged unsuccessful append(\( h \)) response events (i.e., such that the returned value is false). All the following theorems are proven in the supplementary materials.

**Theorem IV.3.** \( \mathcal{R}^0(BT-ADT, \Theta_F) \subseteq \mathcal{R}(BT-ADT, \Theta_P) \).

**Theorem IV.4.** If \( k_1 \leq k_2 \) then \( \mathcal{R}^0(BT-ADT, \Theta_F,k_1) \subseteq \mathcal{R}^0(BT-ADT, \Theta_F,k_2) \).

Finally, from Theorem IV.1, we have the following corollary.

**Corollary IV.4.1.** \( \hat{\mathcal{R}}^0(BT-ADT_{EC};\Theta) \subseteq \hat{\mathcal{R}}^0(BT-ADT_{EC};\Theta) \).

The above results imply the hierarchy depicted in Figure 4. The arrows \( A \to B \) in the figure indicate that the set of histories in \( A \) are included in the set of histories in \( B \) according to Theorems and Lemmas presented in Section V.

### V. Implementing BT-ADTs

#### A. Implementability in the shared memory model

We now consider a system made of \( n \) processes such that up to \( f \) of them are faulty (stop prematurely by crashing), \( f < n \). Non faulty processes are said correct. Processes communicate through atomic registers.

1) Frugal oracle \( \Theta_{F,k=1} \) is at least as strong as Consensus:

We show that there exists a wait-free implementation of Consensus [17] by \( \Theta_{F,k=1} \). Note that similarly to [8], we extend the validity property of Consensus to fit the blockchain setting. Specifically, we have

**Definition V.1** (Consensus \( \mathcal{C} \)).

- **Validity** A value is valid if it satisfies the predefined predicate \( P \).
- **Termination.** Every correct process eventually decides some value, and that value must be valid.
- **Integrity.** No correct process decides twice.
- **Agreement.** If there is a correct process that decides value \( b \), then eventually all the correct processes decide \( b \).

We first show that there exists a wait-free implementation of the Compare&Swap() object by \( \Theta_{F,k=1} \) assuming that blocks are valid, i.e., belong to \( B \). Doing this implies that, under the assumption that blocks are valid, \( \Theta_{F,k=1} \) has the same Consensus number as Compare&Swap(), i.e., \( \infty \) (see [15]). We then show that there is a wait-free implementation of Consensus \( \mathcal{C} \) by \( \Theta_{F,k=1} \) for any block \( b \in B \) (i.e., \( b \) may not be valid). Doing this will imply that \( \Theta_{F,k=1} \) has the same Consensus number as Consensus(), i.e., \( \infty \).
Recall that Compare\&Swap() takes three parameters as input, the register, the old_value and the new_value. If the value in register is the same as old_value then the new_value is stored in register and in any case the operation returns the value that was in register at the beginning of the operation.

Figure 5 proposes an algorithm that reduces CAS object to $\Theta_{F,k=1}$ object.

**Theorem V.1.** If input values are in $B'$ then there exists an implementation of CAS by $\Theta_{F,k=1}$.

Figure 6 describes a simple implementation of Consensus by $\Theta_{F,k=1}$. When a process $p$ invokes procedure Consensus with the block $b_h$ to which $p$ wishes to append its block $b$, it first sets its proposal (Line 1), and then loops invoking the getToken($b_h$, proposal) operation until a valid block is returned (Lines 3-4). Once process $p$ obtain a valid block, it invokes the consumeToken() operation with this valid block as a parameter. The consumeToken() returns the unique valid block for level $level$ (Line 5). Note that this unique valid block is the one of the first process that invoked the consumeToken() operation. Thus the decision value is the valid block of the first process that invoked the consumeToken() operation (see Line 5), and thus it is the same for all the processes.

**Theorem V.2.** $\Theta_{F,k=1}$ Oracle has Consensus number $\infty$.

2) Prodigal oracle $\Theta_P$ is not stronger than Generalized Lattice Agreement: In this section we present a reduction of the prodigal oracle $\Theta_P$ to Generalized Lattice Agreement (GLA) [12]. We will first recall the properties of GLA, a version of lattice agreement generalized to a possibly infinite sequence of input values.

**Definition V.2** (GLA Problem [12]). Let $L$ be a join semi-lattice with a partial order $\sqsubseteq$. Each process may propose an input value belonging to the lattice at any point in time. There is no bound on the number of input values a process may propose. Let $v_i^x$ denote the $x$-th input value proposed by a process $p_i$. The objective is for each process $p_i$ to learn a sequence of output values $w_i^y$ that satisfy the following conditions:

1) **Validity.** Any learnt value $w_i^y$ is a join of some set of input values.
2) **Stability.** The value learnt by any process $p_i$ increases monotonically: $x < y \Rightarrow w_i^x \sqsubseteq w_i^y$. 
3) Consistency. Any two values \( w^i \) and \( w^j \) learnt by any two processes \( p_i \) and \( p_j \) are comparable.

4) Liveness. Every value \( v^i_k \) proposed by a correct process \( p_i \) is eventually included in some learnt value \( w^j_l \) of every correct process \( p_j \), i.e. \( v^i_k \subseteq w^j_l \).

   a) Reduction of the prodigal oracle to Generalized Lattice Agreement: In order to show the reduction of the prodigal oracle to GLA, we consider a lattice for each possible object \( obj_h \), a process wants to append its own object to. Intuitively, in the context of the BT-ADT, the object \( obj_h \) is a vertex of a tree that maps a to a lattice whose input values are subsets of the vertex's children. In order to formally define the input values of the lattice, let us recall that a consume token operation invoked to chain an object \( obj_i \) to a given object \( obj_j \), i.e., \( consumeToken(obj_i^{tkn_h}) \), returns a set of objects that includes the chained object \( obj_j^{tkn_h} \). In this context, the lattice input values belong then to the objects power set, where the greatest lower bound is the empty set.

   Figure 7 shows an implementation of consumeToken by GLA, where the process executes \( proposeValue\{ obj_i^{tkn_h} \} \) of GLA, taking the singleton set \( \{obj_i^{tkn_h}\} \) to be a newly proposed value. The consume token returns a set that reflects all the objects in the learnt set, which includes the proposed object.

\[ \text{validProposal} = \text{consumeToken}(\text{validProposal}) \]

\[ \text{validProposal} = \text{getToken}(h) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]

\[ \text{consumeToken} = \text{consumeToken}(\{h\}) \]
distributed protocol $\mathcal{P}$ it should execute. A non-faulty process is said correct. Processes communicate by exchanging messages over communication channels that can be asynchronous or synchronous (see [6]). We will specify whenever necessary the synchrony assumptions of the channels. By default we consider asynchronous channels.

The BlockTree is considered as a shared object replicated at each process. Let $bt_i$ be the local copy of the BlockTree maintained at process $i$. To maintain the replicated object we consider histories made of events related to the read and append operations on the shared object, i.e. the send and receive operations for process communications and the update operation for BlockTree replica updates. We also use subscript $i$ to indicate that the operation occurred at process $i$: $update_i(b_g, b_t)$ indicates that $i$ inserts its locally generated valid block $b_i$ in $bt_i$ with $b_g$ as a predecessor. Updates are communicated through send and receive operations: an update related to a block $b_t$ generated on a process $p_i$, which is sent through the send$_i(b_g, b_t)$ operation, and which is received through the receive$_j(b_g, b_t)$ operation, takes effect on the local replica $bt_j$ of $p_j$ with the update$_j(b_g, b_t)$ operation.

In the remaining of this work we consider implementations of BT-ADT in a Byzantine failure model where the set of events $E$ is restricted to a countable set of events that contains (i) all the BT-ADT read() operations invocation events by the correct processes, (ii) all BT-ADT read() operations response events at the correct processes, (iii) all append(b) operations invocation events such that $b$ satisfies predicate $P$ and, finally (iv) send, receive and update events generated at correct processes. Note that the Oracle-ADT is by construction agnostic to failures.

1) Necessity of reliable communication: We define the properties on the communication primitive that each history $H$ generated by a BT-ADT satisfying the Eventual Prefix Property must satisfy. We need to first introduce the following definition:

**Definition V.3 (Update agreement).** A concurrent history $H = \langle \Sigma, E, A, \rightarrow, \prec, \nearrow \rangle$ generated by a BT-ADT satisfies the update agreement property if properties R1, R2 and R3 hold.

- **R1.** $\forall update_i(b_g, b_t) \in H, \exists send_i(b_g, b_t) \in H$;
- **R2.** $\forall update_i(b_g, b_t) \in H, \exists receive_i(b_g, b_t) \in H \text{ such that } receive_i(b_g, b_t) \rightarrow update_i(b_g, b_t)$;
- **R3.** $\forall update_i(b_g, b_t) \in H, \exists receive_k(b_g, b_t) \in H, \forall k$.

**Theorem V.4.** The update agreement property is necessary to construct concurrent histories $H = \langle \Sigma, E, A, \rightarrow, \prec, \nearrow \rangle$ generated by a BT-ADT that satisfy the BT Eventual Consistency criterion.

**Proof.** The intuition of the proof is that to meet BT Eventual Consistency all the processes must have the same view of BlockTree eventually. In fact missing an update on the branch that will be eventually selected (which cannot be a-priori-known) would imply that the prefix (which will be arbitrarily long) for the process that missed the update will diverge forever. For space reason the proof of the theorem can be found in the supplementary materials.

We can now present the Light Reliable Communication (LRC) primitive.

**Definition V.4** (Light Reliable Communication (LRC)). A concurrent history $H$ satisfies the properties of the LRC abstraction if and only if:

- (Validity): $\forall send_i(b, b_t) \in H, \exists receive_i(b, b_t) \in H$;
- (Agreement): $\forall receive_i(b, b_j) \in H, \forall k \exists receive_k(b, b_j) \in H$

From Theorem V.4, it is straightforward to show that LRC is necessary to implement BT Eventual consistency (by using arguments from [6]). The proof of the necessity is based on the Validity and Agreement for R1, R2 and R3. The interested reader can refer to the supplementary materials for the proof.

**Theorem V.5.** The LRC primitive is necessary for any BT-ADT implementation that generates concurrent histories which satisfies the BT Eventual Consistency criterion.

By Theorem IV.1, the results trivially hold for the BT Strong consistency criterion.

2) **Impossibility of BT Strong Consistency with forks:** The following theorem states that BT Strong consistency cannot be implemented if forks can occur. Intuitively the proof is based on showing a scenario in which two concurrent updates $b_i$ and $b_j$ are issued, linked to a same block $b$ and two reads at two different processes read $b \sim b_i$ and $b \sim b_j$, violating the Strong prefix property.

**Observation.** Following our Oracle based abstraction (Section IV-C) we assume by definition that the synchronization on the block to append is oracle side and takes place during the append operation. It follows that when an append operation occurs and a correct process updates

---

![Fig. 7: Reduction of the prodigal oracle to Generalized Lattice Agreement](image-url)
its local blocktree then it cannot use anything weaker than the LRC communication abstraction.

**Theorem V.6.** There does not exist an implementation of \( \mathcal{R}(\text{BT-ADT}_\text{SC}, \Theta) \) with \( \Theta \neq \Theta_{F,k=1} \) that uses a LRC primitive and generates histories satisfying the BT Strong consistency.

The non-implementability of refinement \( \mathcal{R}(\text{BT-ADT}_\text{SC}, \Theta_F) \) and \( \mathcal{R}(\text{BT-ADT}_\text{SC}, \Theta_{F,k>1}) \) is a direct implication of the theorem, whose effect is reported in gray in Figure 4.

From Theorem V.6 the next Corollary follows.

**Corollary V.6.1.** \( \Theta_{F,k=1} \) is necessary for any implementation of any \( \mathcal{R}(\text{BT-ADT}_\text{SC}, \Theta) \) that generates histories satisfying the BT Strong consistency.

Thanks to Theorem V.2 the next Corollary also follows.

**Corollary V.6.2.** Consensus is necessary for any implementation of a BT-ADT that generates histories satisfying the BT Strong consistency.

C. **Mapping with existing Blockchain implementations**

We complete this work by illustrating the mapping in the following table between different existing systems and the specifications and abstractions presented in this paper. Interestingly, the mapping shows that all the proposed abstractions are implemented (even though in a probabilistic way in some case), and that the only two refinements used are \( \mathcal{R}(\text{BT-ADT}_\text{SC}, \Theta_{F,k=1}) \) and \( \mathcal{R}(\text{BT-ADT}_\text{EC}, \Theta_F) \). In the following we discuss Bitcoin and Red Belly, an interested reader can find the discussions for the other systems in the supplementary materials.

**TABLE I: Mapping of some existing systems.**

<table>
<thead>
<tr>
<th>References</th>
<th>Refinement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitcoin [19]</td>
<td>( \mathcal{R}(\text{BT-ADT}<em>\text{EC}, \Theta</em>{F,k=1}) ) EC w.h.p</td>
</tr>
<tr>
<td>Ethereum [24]</td>
<td>( \mathcal{R}(\text{BT-ADT}<em>\text{EC}, \Theta</em>{F,k=1}) ) EC w.h.p</td>
</tr>
<tr>
<td>Algorand [13]</td>
<td>( \mathcal{R}(\text{BT-ADT}<em>\text{SC}, \Theta</em>{F,k=1}) ) SC w.h.p</td>
</tr>
<tr>
<td>ByzCoin [16]</td>
<td>( \mathcal{R}(\text{BT-ADT}<em>\text{SC}, \Theta</em>{F,k=1}) )</td>
</tr>
<tr>
<td>PeerCensus [9]</td>
<td>( \mathcal{R}(\text{BT-ADT}<em>\text{SC}, \Theta</em>{F,k=1}) )</td>
</tr>
<tr>
<td>Redbelly [8]</td>
<td>( \mathcal{R}(\text{BT-ADT}<em>\text{SC}, \Theta</em>{F,k=1}) )</td>
</tr>
<tr>
<td>Hyperledger [4]</td>
<td>( \mathcal{R}(\text{BT-ADT}<em>\text{SC}, \Theta</em>{F,k=1}) )</td>
</tr>
</tbody>
</table>

D. **Bitcoin**

In Bitcoin [19] each process \( p \in V \) is allowed to read the BlockTree and append blocks to the BlockTree. Processes are characterized by their computational power represented by \( \alpha_p \), normalized as \( \sum_{p \in V} \alpha_p = 1 \). Processes communicate through reliable FIFO authenticated channels, which models a partially synchronous setting [10]. Valid blocks are flooded in the system. The getToken operation is implemented by a proof-of-work mechanism. The consumeToken operation returns true for all valid blocks, thus there is no bounds on the number of consumed tokens. Thus Bitcoin implements a Prodigal Oracle. The selection function \( f \) selects the blockchain which has required the most computational work, guaranteeing that concurrent blocks can only refer to the most recently appended blocks of the blockchain returned by a read() operation. Garay and al [20] have shown, under a synchronous environment assumption, that Bitcoin ensures Eventual consistency criteria with high probability. The same conclusion applies as well for the FruitChain protocol [21], which proposes a protocol similar to BitCoin except for the rewarding mechanism.

E. **Red Belly**

Red Belly [8] is a consortium blockchain, meaning that any process \( p \in V \) is allowed to read the BlockTree but a predefined subset \( M \subseteq V \) of processes are allowed to append blocks. Each process \( p \in M \) has a merit parameter set to \( \alpha_p = 1/|M| \) while each process \( p \in V \setminus M \) has a merit parameter \( \alpha_p = 0 \). Each process \( p \in M \) can invoke the getToken operation with their new block and will receive a token. The consumeToken operation, implemented by a Byzantine consensus algorithm run by all the processes in \( V \), returns true for the uniquely decided block. Thus Red Belly BlockTree contains a unique blockchain, meaning that the selection function \( f \) is the trivial projection function from \( BT \rightarrow BC \) which associates to the BT-ADT its unique existing chain of the BlockTree. As a consequence Red Belly relies on a Frugal Oracle with \( k = 1 \), and by the properties of Byzantine agreement implements a strongly consistent BlockTree (see Theorem 3 [8]).

VI. **Conclusions and Future Work**

The paper presented a formal specification of blockchains and derived interesting conclusions on their implementability. Let us note that the presented work is intended to provide the groundwork for the construction of a sound hierarchy of blockchain abstractions and correct implementations. We believe that the presented results are also of practical interests since our oracle construction not only reflects the design of many current implementations, but will help designers in choosing the oracle they want to implement with a clear semantics and inherent trade-offs in mind. Future work will focus on several open issues, such as the solvability of Eventual Prefix in message-passing, the synchronization power of other oracle models, and fairness properties for oracles.

a) **Acknowledgments**: The authors thank the referees for their helpful comments.

**REFERENCES**


