Blockchain abstract data type

Emmanuelle Anceaume, Antonella del Pozzo, Romaric Ludinard, Maria Potop-Butucaru, Sara Tucci-Piergiovanni

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 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 *hierachy 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 ∞ 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 $T = \langle A, B, Z, \xi_0, \tau, \delta \rangle$. 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 τ. 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 δ. 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 Λ is the function which associates events to the operations in Σ. Given two events $(e, e') \in E^2$ we say that $e \mapsto e'$ 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 $e \nearrow e'$, if $e \mapsto e'$ or $e \prec e'$. These asymmetric event structures allow us to define a concurrent history of an ADT T =$\langle A, B, Z, \xi_0, \tau, \delta \rangle$ as a 6-tuple $H = \langle \Sigma, E, \Lambda, \mapsto, \prec, \nearrow \rangle$ [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 $\mathsf{append}(b_\ell)$ operation, which appends a new block b_ℓ to the BlockTree, and the $\mathsf{read}()$ 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 $\perp$) 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 $\mathcal{B}$ a countable and non empty set of blocks and by $\mathcal{B}' \subseteq \mathcal{B}$ a countable and non empty set of valid blocks, i.e., $\forall b \in \mathcal{B}'$, $P(b) = \top$. By assumption $b_0 \in \mathcal{B}'$; We also denote by $\mathcal{BC}$ a countable non empty set of blockchains, where a blockchain is a path from a leaf of bt to b_0. A blockchain is denoted by bc. Finally, $\mathcal{F}$ is a countable non empty set of selection functions, $f \in \mathcal{F} : \mathcal{BT} \to \mathcal{BC}$; $f(bt)$ 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: $\{b_0\}^\frown f(bt)$ represents the concatenation of b_0 with the blockchain of bt; and $\{b_0\}^\frown f(bt)^\frown \{b\}$ 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 ($BT\text{-}ADT$))**.** The BlockTree Abstract Data Type is the 6-tuple BT-ADT=$\langle A = \{\mathsf{append}(b_h, b_\ell), \mathsf{read}()\colon b \in \mathcal{B}\}, B = \mathcal{BC} \cup \{\mathsf{true}, \mathsf{false}\}, Z = \mathcal{BT} \times \mathcal{F} \times (\mathcal{B} \to \{\mathsf{true}, \mathsf{false}\}),\ \xi_0 = (b_0, f, P), \tau, \delta\rangle$, where the transition function $\tau : Z \times A \to Z$ is defined by

$$\tau((bt, f, P), \mathsf{read}()) = (bt, f, P)$$
$$\tau((bt, f, P), \mathsf{append}(b)) = \begin{cases} (\{b_0\}^\frown f(bt)^\frown \{b\}, f, P) \text{ if } b \in \mathcal{B}' \\ (bt, f, P) \text{ otherwise} \end{cases}$$

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

$$\delta((bt, f, P), \mathsf{read}()) = \begin{cases} \{b_0\} \text{ if } bt = b_0 \\ \{b_0\}^\frown f(bt) \text{ otherwise} \end{cases}$$
$$\delta((bt, f, P), \mathsf{append}(b)) = \begin{cases} \mathsf{true} \text{ if } b \in \mathcal{B}' \\ \mathsf{false} \text{ otherwise} \end{cases}$$

The semantic of the read and the append operations directly depends on the selection function $f \in \mathcal{F}$. 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 ξ_0, state ξ_1 is obtained by appending block b_1 to ξ_0 and state ξ_2 is obtained by appending block b_2 to ξ_1. The read operation applied in state ξ_1 returns blockchain $\{b_0\}^\frown\{b_1\}$, and the read applied in state ξ_2 returns blockchain $\{b_0\}^\frown f(bt)^\frown\{b_2\} = \{b_0\}^\frown\{b_1\}^\frown\{b_2\}$.

2) Concurrent histories of a BT-ADT and consistency criteria: A $BT\text{-}ADT$ 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 $\mathsf{append}()$ and $\mathsf{read}()$ invocation and response events. Similarly, $E(a, r^*)$ refers to an infinite set containing *(i)* a finite number of $\mathsf{append}()$ invocation and response events and *(ii)* an infinite number of $\mathsf{read}()$ invocation and response events;
- $\mathsf{score} : \mathcal{BC} \to \mathbb{N}$ 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. $\mathsf{score}(\{b_0\}) = s_0$. Increasing monotonicity means that $\mathsf{score}(bc^\frown\{b\}) > \mathsf{score}(bc)$;
- $\mathsf{mcps} : \mathcal{BC} \times \mathcal{BC} \to \mathbb{N}$ 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 $\mathsf{read}()$ 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 $\mathsf{read}()$ operation is *valid* (i.e.,

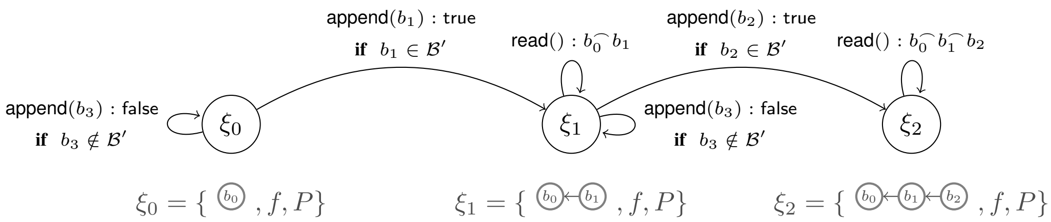

Fig. 1: A possible path of the transition system defined by the BT-ADT.

satisfies predicate P) and has previously been inserted in the BlockTree with the append() operation. Formally,

Definition IV.2 (Block validity)**.** $\forall e_{rsp}(r) \in E, \forall b \in e_{rsp}(r) : bc, b \in \mathcal{B}' \wedge \exists e_{inv}(\mathsf{append}(b)) \in E, e_{inv}(append(b)) \nearrow e_{rsp}(r)$

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

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

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)**.** $\forall e_{rsp}(r), e_{rsp}(r') \in E^2, (e_{rsp}(r') : bc' \sqsubseteq e_{rsp}(r) : bc) \vee (e_{rsp}(r) : bc \sqsubseteq e_{rsp}(r') : bc')$

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 read() operation r, the set of read() operations r' such that $e_{rsp}(r) \nearrow e_{inv}(r')$ that do not return blockchains with a score greater than s is finite. Formally,

Definition IV.5 (Ever growing tree)**.** $\forall e_{rsp}(r) \in E(a^*, r^*)$, $s = \mathsf{score}(e_{rsp}(r) : bc)$ then
$|\{e_{inv}(r') \in E \mid e_{rsp}(r) \nearrow e_{inv}(r'), \mathsf{score}(e_{rsp}(r') : bc') \leq s\}| < \infty$

Definition IV.6 (BT Strong Consistency (SC) criterion)**.** A concurrent history $H = \langle \Sigma, E, \Lambda, \mapsto, \prec, \nearrow \rangle$ 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 read() 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 read() operation with s as score, then eventually all the read() 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 read() operations, and let s be the score of the blockchain returned by a read() operation r then, the set of read() operations r', such that $e_{rsp}(r) \nearrow e_{inv}(r')$, 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 $H = \langle \Sigma, E(a, r^*), \Lambda, \mapsto, \prec, \nearrow \rangle$ of the system that uses a BT-ADT, we denote by s, for any read operation $r \in \Sigma$ such that $\exists e \in E(a, r^*), \Lambda(r) = e$, the score of the returned blockchain, *i.e.*, $s = \mathsf{score}(e_{rsp}(r) : bc)$. We denote by E_r the set of response events of read operations that occurred after r response, *i.e.* $E_r = \{e \in E \mid \exists r' \in \Sigma, r' = \mathsf{read}, e = e_{rsp}(r') \wedge e_{rsp}(r) \nearrow e_{rsp}(r')\}$. Then, H satisfies the Eventual prefix property if for all read() operations $r \in \Sigma$ with score s, there is a set $S = \{(e_{rsp}(r_h), e_{rsp}(r_k)) \in E_r^2 | h \neq k, \mathsf{mcps}(e_{rsp}(r_h) : bc_h, e_{rsp}(r_k) : bc_k) < s\}$ and $|S| < \infty$.

Definition IV.8 (BT Eventual Consistency (EC) criterion)**.** A concurrent history $H = \langle \Sigma, E, \Lambda, \mapsto, \prec, \nearrow \rangle$ 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 $\mathcal{H}_{EC}$ and $\mathcal{H}_{SC}$ 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., $\mathcal{H}_{SC} \subset \mathcal{H}_{EC}$.

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 Θ

We now formalize the Token Oracle Θ 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 b_ℓ to $b_h \in \mathcal{B}'$, if it successfully gains a token tkn_h from the token oracle Θ. Once obtained, the proposed block b_ℓ is considered as valid, and will be denoted by $b_\ell^{tkn_h}$. By construction $b_\ell^{tkn_h} \in \mathcal{B}'$. 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_h, i.e., $P(obj_q) = \top$ it invokes the getToken(obj_h, obj_ℓ) operation with object obj_ℓ from set $\mathcal{O} = \{obj_1, obj_2, \dots\}$. If getToken$(obj_h, obj_\ell)$ operation is successful, it returns the valid object $obj_\ell^{tkn_h}$ such that tkn_h is the token required to access valid object obj_h. The set of valid objects is denoted by $\mathcal{O}'$, i.e., $\forall obj_q \in \mathcal{O}', P(obj_q) = \top$. We say that a valid object is *generated* each time it is successfully returned by a getToken(obj_h, obj_ℓ) operation and it is *consumed* when the oracle grants the right to associate this valid object $obj_\ell^{tkn_h}$ to obj_h. In the following, once an object is valid, if it is clear from the context, we will not explicit the token tkn with makes the object valid.

A valid object $obj_\ell^{tkn_h}$ is consumed through the consumeToken$(obj_\ell^{tkn_h})$ operation. No more than k valid objects $obj_{\ell_1}^{tkn_h}, \dots, obj_{\ell_j}^{tkn_h}$, $1 \leq j \leq k$, can be consumed for obj_h, where k is a parameter of the token oracle. The side-effect of the consumeToken$(obj_\ell^{tkn_h})$ on the state of the token oracle is the insertion of the valid object $obj_\ell^{tkn_h}$ in a set related to obj_h 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 Θ_P, has no upper bound on the number of tokens consumed for an object, while the second oracle Θ_F, called *frugal*, and denoted by Θ_F, guarantees that no more than k token can be consumed for each object.

The prodigal oracle Θ_P when combined with the BlockTree abstract data type will only help in validating blocks, while the frugal oracle Θ_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_h, obj_ℓ) operation is invoked, the oracle provides a valid object with a certain probability $p_{\alpha_i} > 0$ where α_i is a "merit" parameter characterizing the invoking process i. [1] Note that the oracle knows α_i of the invoking process i, which might be unknown to the process itself. For each merit α_i, the state of the token oracle embeds an infinite tape where each cell of the tape contains either tkn or $\perp$. Since each tape is identified by a specific α_i and p_{α_i}, we assume that each tape contains a pseudorandom sequence of values in $\{tkn, \perp\}$ depending on α_i. [2]

When a getToken(obj_h, obj_ℓ) operation is invoked by a process with merit α_i, the oracle pops the first cell from the tape associated to α_i, and a valid obeject 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_h, which is populated each time a valid object obj_ℓ is consumed. When the set cardinality reaches k then no more tokens can be consumed for that object. For the sake of generality, Θ_P is defined as Θ_F with $k = \infty$ while for Θ_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 $\mathcal{O}'$, *(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 Θ_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 Θ_P-ADT. When a process with merit α_1 invokes getToken(b_1, b_k), with b_1 the leaf of $f(bt)$), the first cell of $tape_{\alpha_1}$ is popped, and if it contains a token, then getToken(b_1, b_k) returns a valid block $b_k^{tkn_1}$. Afterwards, when consumeToken$(b_k^{tkn_1})$ is invoked, the oracle checks if the cardinality of the set in $K[1]$ is strictly smaller than k, and if the affirmative inserts $b_k^{tkn_1}$ in $K[1]$. In any cases, consumeToken$()$ returns the content of $K[1]$, in this case $b_k^{tkn_1}$. It follows that a process that gets a valid block for some block b_h 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 Θ Oracles

We augment the BT-ADT with Θ oracles and we analyze the histories generated by their combination. Specifically, we define a refinement of the append(b_ℓ) operation of the BT-ADT with the oracle operations as follows: the append(b_ℓ) operation triggers the getToken$(b_h \leftarrow$last_block$(f(bt)), b_\ell)$ operation as long as it returns a valid block ${b_\ell}^{tkn_h}$, and once obtained, the valid block might be consumed, and in any cases the append(b_ℓ) 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 $X1, X2, X3, \dots$ such that *(i)* for each k, the value of X_k is either tkn or $\perp$; and *(ii)* $\forall X_k$ the probability that $X_k = tkn$ is p_{α_i}.

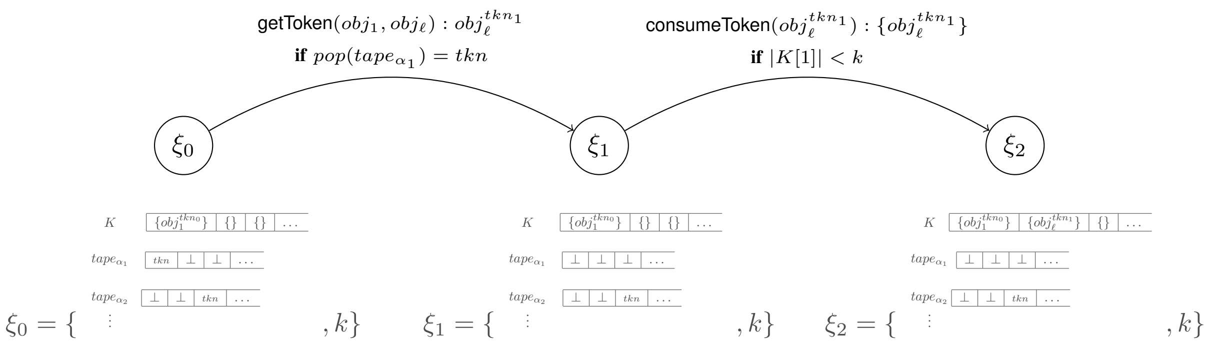

Fig. 2: A possible path of the transition system defined by the Θ_F and $\Theta_{F,k}$-ADTs.

blocks have already been consumed for b_h, the valid block is consumed i.e. block $b_\ell{}^{tkn_h}$ is appended to the block h in the blockchain $f(bt)$ (i.e., $\{b_0\}^\frown f(bt)|_h^\frown\{b_\ell\}$) and the append($b_\ell$) operation returns true, otherwise false. We say that the $BT\text{-}ADT$ augmented with Θ_F or Θ_P oracle is a *refinement* $\mathfrak{R}(BT\text{-}ADT, \Theta_F)$ or $\mathfrak{R}(BT\text{-}ADT, \Theta_P)$ respectively. The formal specification of these refinements are given in the supplementary materials.

Definition IV.9 (k-Fork coherence)**.** A concurrent history $H = \langle \Sigma, E, \Lambda, \mapsto, \prec, \nearrow \rangle$ of $\mathfrak{R}(BT\text{-}ADT, \Theta_{F,k})$ satisfies the *k-Fork coherence* if there are at most k append($b_\ell^{tkn_h}$) operations that return true for the same block b_ℓ.

Theorem IV.2 (k-Fork coherence)**.** Any concurrent history $H = \langle \Sigma, E, \Lambda, \mapsto, \prec, \nearrow \rangle$ of the $\mathfrak{R}(BT\text{-}ADT, \Theta_{F,k})$ satisfies the *k-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: $\mathfrak{R}$(BT-ADT$_{SC}, \Theta_{F,k}$), $\mathfrak{R}$(BT-ADT$_{SC}, \Theta_P$), $\mathfrak{R}$(BT-ADT$_{EC}, \Theta_P$), $\mathfrak{R}$(BT-ADT$_{EC}, \Theta_{F,k}$). We aim at studying the relationships among the different refinements. Let $\hat{\mathcal{H}}^{\mathfrak{R}(\text{BT-ADT},\Theta_{F,k})}$ be the set of concurrent histories generated by a BT-ADT enriched with $\Theta_{F,k}$-ADT and $\hat{\mathcal{H}}^{\mathfrak{R}(\text{BT-ADT},\Theta_P)}$ be the set of concurrent histories generated by a BT-ADT enriched with Θ_P-ADT. Without loss of generality, we consider only the set of histories from which have been purged unsuccessful append() response events (i.e., such that the returned value is false). All the following theorems are proven in the supplementary materials.

Theorem IV.3. $\hat{\mathcal{H}}^{\mathfrak{R}(\text{BT-ADT},\Theta_F)} \subseteq \hat{\mathcal{H}}^{\mathfrak{R}(\text{BT-ADT},\Theta_P)}$.

Theorem IV.4. If $k_1 \leq k_2$ then $\hat{\mathcal{H}}^{\mathfrak{R}(\text{BT-ADT},\Theta_{F,k_1})} \subseteq \hat{\mathcal{H}}^{\mathfrak{R}(\text{BT-ADT},\Theta_{F,k_2})}$.

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

Corollary IV.4.1. $\hat{\mathcal{H}}^{(\mathfrak{R}(\text{BT-ADT}_{SC},\Theta)} \subseteq \hat{\mathcal{H}}^{\mathfrak{R}(\text{BT-ADT}_{EC},\Theta)}$.

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 $\mathcal{B}'$. 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., ∞ (see [15]). We then show that there is a wait-free implementation of Consensus $\mathcal{C}$ by $\Theta_{F,k=1}$ for any block $b \in \mathcal{B}$ (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., ∞.

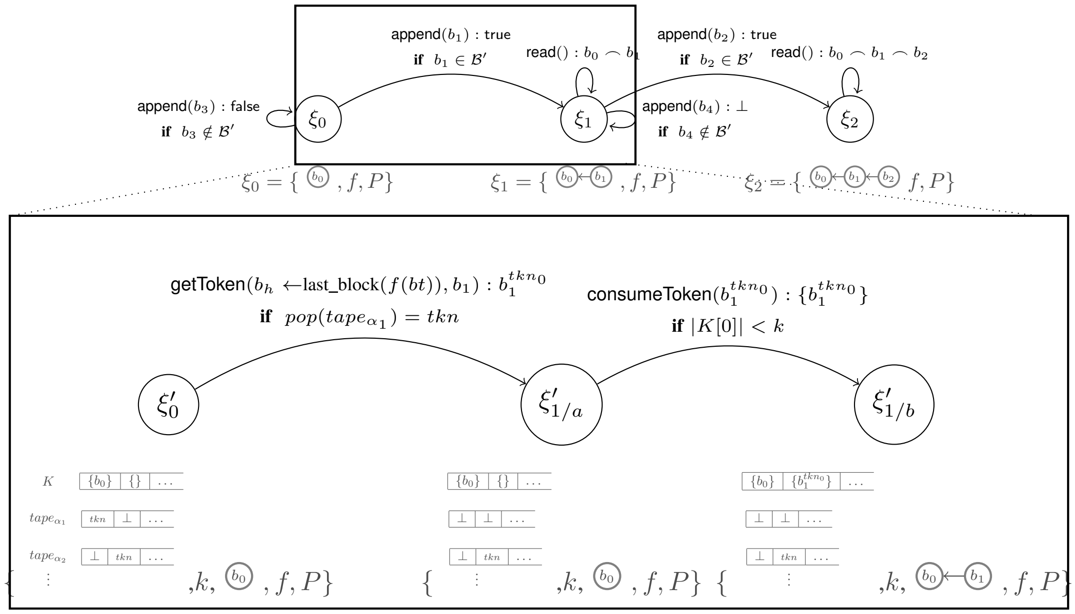

Fig. 3: A possible path of the transition system defined by the refinement of the append() operation.

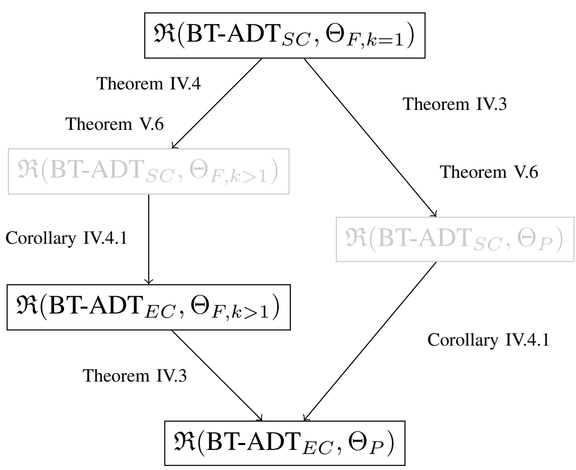

Fig. 4: $\Re(\text{BT-ADT}, \Theta)$ Hierarchy. In gray we anticipate the combinations impossible in a message-passing system due to Theorem V.6.

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 $\mathcal{B}'$ 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 ∞.

2) Prodigal oracle Θ_P is not stronger than Generalized Lattice Agreement: In this section we present a reduction of the prodigal oracle Θ_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 : $x < y \Rightarrow w_i^x \sqsubseteq w_i^y$.

```
(1) compare&swap(K[h], {}, b_ℓ^{tkn_h}) :
(2) first ← consumeToken(b_ℓ^{tkn_h});
(3) if (first == b_ℓ^{tkn_h})
(4)     then previous_value = {};
(5)     else previous_value = first;
(6) endIf
(7) return previous_value;
```

Fig. 5: An implementation of CAS by the Frugal Oracle with $k = 1$.

```
Consenus(b_h):
(1) proposal ← b;
(2) validProposal ← ⊥;
(3) while (validProposal = ⊥):
(4)     validProposal ← getToken(b_h, proposal);
(5) return (consumeToken(validProposal));
```

Fig. 6: An implementation of Consensus by the Frugal Oracle with $k = 1$.

3) **Consistency.** Any two values w_i^x and w_j^y learnt by any two processes p_i and p_j are comparable.
4) **Liveness.** Every value v_i^x proposed by a correct process p_i is eventually included in some learnt value w_j^y of every correct process p_j, i.e. $v_i^x \sqsubseteq w_j^y$

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 to a lattice whose input values are subsets of the vertexś 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_ℓ to a given object obj_h, i.e., consumeToken($obj_\ell^{tkn_h}$), returns a set of objects that includes the chained object $obj_\ell^{tkn_h}$. 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($\{obj_\ell^{tkn_h}\}$) of GLA, taking the singleton set $\{obj_\ell^{tkn_h}\}$ 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.

Theorem V.3. Θ_P Oracle is not stronger than Generalized Lattice Agreement.

Proof. (Sketch)

The proof follows from the implementation in Figure 7. Let us recall that the oracle must behave as an atomic object, which means that we need to show that the oracle is linearizable through GLA. GLA proposed values in our implementation are sets, where each proposed value is a singleton set containing a uniquely identified object. The join of any two proposed values is the union of the proposed singleton sets. Any learnt set is the union of some proposed sets. Any two learnt sets are comparable through the inclusion operator. The first step is to show that the order of non-overlapping consumeToken operations is preserved: if a process p_i completes a consumeToken ct_1 operation before another process p_j invokes another ct_2 operation, then we must ensure that ct_1 occurs before ct_2 in the linearization order, i.e. the effect of ct_1 is visible to ct_2. Note that from the pseudo-code, the only values included in $K[h]$ are learnt values, i.e. a join of some proposed values by the GLA Validity and from Line 2. Moreover, from Line 3 each process waits for its own proposed set to be learnt before the consumeToken completes. This means that the proposed set set_1 by ct_1 is learnt and included in $K[h]$, before ct_2 is invoked. Since the learnt value set_1 through ct_1 must now be comparable to the learnt set set_2 through ct_2, this implies that the learnt set set_2 through ct_2 must also include set_1. $K[h]$ will then include set_1, i.e. ct_2 has seen the effect of ct_1. The second step is to show that any two concurrent operations ct_1 and ct_2 can be linearized. By Consistency, even in this case the learnt values must be comparable, either set_1 is included in set_2 or the other way round. In both cases the effect of one operation is visible to the other one, and then they can be linearized. The last step is to show the the implementation is wait-free. Wait-freedom is ensured by the Liveness property of GLA that ensures that the execution time of Line 3 is finite. □

B. Implementability in a message-passing system model

In this section we are interested in distributed message-passing implementations of BT-ADTs. In the following, we will present *(i)* the necessity of a light form of reliable broadcast to implement BT Eventual consistency, *(ii)* refinement of BTs with Oracles that are not implementable in a message-passing system and *(iii)* the mapping of current existing implementations with our abstract data types.

To this end, we consider a message-passing system composed of an arbitrarily large but finite set of n processes, such that a subset of them can fail by exhibiting Byzantine failures, that is deviates arbitrarily from the

```
consumeToken(obj_ℓ^{tkn_h})
    proposeValue({obj_ℓ^{tkn_h}})
    wait until obj_ℓ^{tkn_h} ∈ LearntValue()
    K[h] = K[h]∪ LearntValue()
    return K[h];
```

Fig. 7: Reduction of the prodigal oracle to Generalized Lattice Agreement

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: $\mathsf{update}_i(b_g, b_\ell)$ indicates that i inserts its locally generated valid block b_ℓ in bt_i with b_g as a predecessor. Updates are communicated through send and receive operations: an update related to a block b_ℓ generated on a process p_i, which is sent through the $\mathsf{send}_i(b_g, b_\ell)$ operation, and which is received through the $\mathsf{receive}_j(b_g, b_\ell)$ operation, takes effect on the local replica bt_j of p_j with the $\mathsf{update}_j(b_g, b_\ell)$ 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 $H = \langle \Sigma, E, \Lambda, \mapsto, \prec, \nearrow \rangle$ generated by a BT-ADT satisfies the update agreement property if properties R1, R2 and R3 hold.

- R1. $\forall \mathsf{update}_i(b_g, b_\ell) \in H$, $\exists \mathsf{send}_i(b_g, b_\ell) \in H$;
- R2. $\forall \mathsf{update}_i(b_g, b_\ell) \in H, \exists \mathsf{receive}_i(b_g, b_\ell) \in H$ such that $\mathsf{receive}_i(b_g, b_\ell) \mapsto \mathsf{update}_i(b_g, b_\ell)$;
- R3. $\forall \mathsf{update}_i(b_g, b_\ell) \in H$, $\exists \mathsf{receive}_k(b_g, b_\ell) \in H, \forall k$.

Theorem V.4. The update agreement property is necessary to construct concurrent histories $H = \langle \Sigma, E, \Lambda, \mapsto, \prec, \nearrow \rangle$ 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): $\forall \mathsf{send}_i(b, b_i) \in H, \exists \mathsf{receive}_i(b, b_i) \in H$;
- (Agreement): $\forall \mathsf{receive}_i(b, b_j) \in H, \forall k \exists \mathsf{receive}_k(b, b_j) \in H$

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 $b^\frown b_i$ and $b^\frown b_j$, 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 its local blocktree then it cannot use anything weaker than the LRC communication abstraction.

Theorem V.6. There does not exist an implementation of $\mathfrak{R}(\text{BT-ADT}_{SC}, \Theta)$ 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 $\mathfrak{R}(\text{BT-ADT}_{SC}, \Theta_P)$ and $\mathfrak{R}(\text{BT-ADT}_{SC}, \Theta_{F,k>1})$ 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 $\mathfrak{R}(\text{BT-ADT}_{SC}, \Theta)$ 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 $\mathfrak{R}(BT\text{-}ADT_{SC}, \Theta_{F,k=1})$ and $\mathfrak{R}(BT\text{-}ADT_{EC}, \Theta_P)$. In the following we discuss Bitcoin and Redbelly, an interested reader can find the discussions for the other systems in the supplementary materials.

TABLE I: Mapping of some existing systems.

References	Refinement
Bitcoin [19]	$\mathfrak{R}(BT\text{-}ADT_{EC}, \Theta_P)$ EC w.h.p
Ethereum [24]	$\mathfrak{R}(BT\text{-}ADT_{EC}, \Theta_P)$ EC w.h.p
Algorand [13]	$\mathfrak{R}(BT\text{-}ADT_{SC}, \Theta_{F,k=1})$ SC w.h.p
ByzCoin [16]	$\mathfrak{R}(BT\text{-}ADT_{SC}, \Theta_{F,k=1})$
PeerCensus [9]	$\mathfrak{R}(BT\text{-}ADT_{SC}, \Theta_{F,k=1})$
Redbelly [8]	$\mathfrak{R}(BT\text{-}ADT_{SC}, \Theta_{F,k=1})$
Hyperledger [4]	$\mathfrak{R}(BT\text{-}ADT_{SC}, \Theta_{F,k=1})$

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 α_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 $\mathcal{BT} \mapsto \mathcal{BC}$ 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

[1] I. Abraham and D. Malkhi. The blockchain consensus layer and BFT. *Bulletin of the EATCS*, 3(123):1–23, 2017.
[2] E. Anceaume, R. Ludinard, M. Potop-Butucaru, and F. Tronel. Bitcoin a distributed shared register. In *Proceedings of the International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, 2017.
[3] E. Anceaume, A. D. Pozzo, R. Ludinard, M. Potop-Butucaru, and S. Tucci Piergiovanni. Blockchain abstract data type - full version. https://hal.archives-ouvertes.fr/hal-02113770, 2019.

[4] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. D. Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolic, S. W. Cocco, and J. Yellick. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. https://arxiv.org/pdf/1801.10228v1.pdf, 2018.

[5] A. F. Anta, K. Konwar, C. Georgiou, and N. Nicolaou. Formalizing and implementing distributed ledger objects. *ACM SIGACT News*, 49(2):58–76, 2018.

[6] C. Cachin, R. Guerraoui, and L. E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011.

[7] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and efficient asynchronous broadcast protocols. In *Proceedings of the Annual International Cryptology Conference on Advances in Cryptology (CRYPTO)*, 2001.

[8] T. Crain, V. Gramoli, M. Larrea, and M. Raynal. (Leader/Randomization/Signature)-free Byzantine Consensus for Consortium Blockchains. http://arxiv.org/abs/1702.03068, 2017.

[9] C. Decker, J. Seidel, and R. Wattenhofer. Bitcoin meets strong consistency. In *Proceedings of the 17th International Conference on Distributed Computing and Networking*, page 13, New York, NY, USA, 2016. ACM.

[10] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in presence of partial synchrony. *Journal of the ACM (JACM)*, 1988.

[11] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In *Advances in Cryptology - CRYPTO '92, 12th Annual International Cryptology Conference, Santa Barbara, California, USA, August 16-20, 1992, Proceedings*, pages 139–147, 1992.

[12] J. M. Falerio, S. Rajamani, K. Rajan, G. Ramalingam, and K. Vaswani. Generalized lattice agreement. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 2012.

[13] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 51–68, New York, NY, USA, 2017. ACM.

[14] A. Girault, G. Gössler, R. Guerraoui, J. Hamza, and D.-A. Seredinschi. Monotonic prefix consistency in distributed systems. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, pages 41–57, Berlin, Germany, 2018. Springer.

[15] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[16] E. Kokoris-Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 279–296, Berkeley, CA, USA, 2016. USENIX Association.

[17] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[18] D. Mazieres and D. Shasha. Building secure file systems out of byzantine storage. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 108–117. ACM, 2002.

[19] S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/bitcoin.pdf, 2008.

[20] E. Oswald and M. Fischlin, editors. *The Bitcoin Backbone Protocol: Analysis and Applications*, volume 9057 of *Lecture Notes in Computer Science*. Springer, 2015.

[21] R. Pass and E. Shi. Fruitchains: A fair blockchain. In *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017*, pages 315–324, New York, NY, USA, 2017. ACM.

[22] M. Perrin. *Distributed Systems, Concurrency and Consistency*. ISTE Press, Elsevier, 2017.

[23] M. Perrin, A. Mostéfaoui, and C. Jard. Causal consistency: beyond memory. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*, pages 26:1–26:12, New York, NY, USA, 2016. ACM.

[24] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. http://gavwood.com/Paper.pdf, 2014.