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

Blockchains (e.g. Bitcoin, Algorand, Byzcoin, Hyperledger, RedBelly etc) became a game changer in the distributed storage area due to their ability to mimic the functioning of a classical traditional ledger such as transparency and falsification-proof of documentation in an untrusted environment where the computation is distributed, the set of participants to the system are not known and it varies during the execution. However, the massive integration of distributed ledgers in industrial applications strongly depends on the formal guaranties of the quality of services offered by these applications, especially in terms of consistency. Our work continues the line of recent distributed computing community effort dedicated to the theoretical aspects of blockchains. This paper is the first to specify the distributed shared ledgers as a composition of *abstract data types* all together with an hierarchy of *consistency criteria* that formally characterizes the histories admissible for distributed programs that use them. Our work extends the consistency criteria theory with new consistency definitions that capture the eventual convergence process in blockchain systems. Furthermore, we map representative existing blockchains from both academia and industry in our framework. Finally, we identify the necessary communication conditions in order to implement the new defined consistency criteria.

1 Introduction

Blockchain technology became today one of the most appealing area of research motivated mainly by its huge potential in the design of a broad class of secured distributed applications. Blockchain systems maintain a continuously-growing history of ordered blocks, each block being composed of transactions that have been verified by the members of the system, called miners. Blocks are linked to each other by relying on hash functions and the order of blocks in the blockchain is the result of a form of agreement among the system participants. At any time, participants strongly agree on a prefix of the blockchain, the suffix of the blockchain being potentially different at participants.

We are interested in formalizing the blockchains as a family of formal specifications that can be combined to meet specific consistency criteria. To this end we provide specifications as a composition of *abstract data types* (whose definition is recalled in Section 2) together with a hierarchy of *consistency criteria* that formally characterizes the histories admissible for distributed programs that use them. The advantage of specifying shared objects as abstract data types over implementation-based alternatives such as [11] is the possibility to reason on the consistency of a system independently of the communication model [17]. More precisely, we define two abstract data types, the *BlockTree* and the *Oracle*. The BlockTree models the blockchain data structure, which is a tree of blocks providing an *append* and a *read* operation. The append operation is intended to insert a new leaf in the tree, provided that the newly inserted block is valid. The validity property of a block in the BlockTree is abstracted as a general predicate, which is application dependent. For instance, in Bitcoin a valid block is a block containing a hash hs of three fields: a set of non-double spending transactions, the hash of the previous block, and a nonce such that hs starts with a given number of leading zeros.

In our formalisation, what makes a block valid, i.e. triggering the validity predicate at true, is indeed the *Oracle*. The Oracle abstracts the mechanism used to grant processes the insertion of new blocks in the blockchain, such as proof-of-work or other agreement mechanisms. To this end the Oracle is modeled as a simple token manager. We say that a token is generated when a process obtains a valid block and it is consumed when it obtains the right to insert it in the chain. Those notions are presented in Section 3.

Let us note that it would have been possible to abstract a blockchain only through the BlockTree abstract data type, including the validation process in the append operation. We advocate that the separation of the blockchain in two different abstractions, however, has several benefits. The first advantage is the possibility to deal with liveness properties at finer granularity level, separating the termination of the validation process, managed by the Oracle, from the termination of updating a replicated data structure, managed by the BlockTree. That separation is very useful since in many implementations the proof-of-work mechanism is a probabilistic local process, while the process of updating a replicated data structure relies on reliable update diffusion and some local deterministic rule to eventually converge on the same totally ordered set of updates. As for safety properties, the separation allows to properly deal with the notion of validity. As it has been noted in [5] a valid block indeed is always "admissible", even if sent by a byzantine process. This notion of validity can then be captured to properly define the problem of updating a replicated data structure with valid data, solely managed by the BlockTree. Furthermore, the possibility to reason on the BlockTree in isolation, allows us to extend the consistency criteria theory with a new consistency criterion for BlockTree that captures the eventual convergence process in blockchain systems. This criterion, presented in Section 3.1 is a weaker consistency criterion than the so-called monotonic prefix consistency criterion, introduced in [11]. The monotonic prefix consistency criterion informally says that any two reads return two chains such that one is the prefix of the other. We indeed relax this consistency criterion admitting any two chains to have a divergent prefix for a finite interval of the history, defining then the *Eventual Prefix* property. Note that we also define a so-called *Strong Prefix* property that is equivalent, under certain conditions, to the monotonic prefix consistency criterion. In the end, we define two BlockTree versions, one enjoying the Eventual Prefix property and the other one enjoying the Strong Prefix property.

The oracle enjoys two different versions as well. The weakest one is an oracle "without memory", called *Prodigal* and the other is an oracle "with memory", called *Frugal*. The Prodigal Oracle does not remember how many tokens are consumed for a given block to extend, while the Frugal oracle counts the number of tokens actually consumed for any block. In practical terms, the Prodigal oracle does not control the number of forks in the system, while the Frugal oracle embeds mechanisms to restrict the number of forks up to k.

Given those specifications we formally show in Section 4.2 the necessary conditions that must hold on the communication model to implement the BlockTree with Eventual Prefix in a message-passing system. Interestingly, we show the necessity of a light form of reliable broadcast enjoying both the Validity (if a correct process sends a message then it eventually delivers it) and Agreement (if a message is delivered by some correct process then the message is eventually delivered by every correct process) properties.

We finally map in Section 5 our specifications to existing representative blockchains. We show that Bitcoin and Ethereum implement a prodigal oracle and their executions are eventually consistent, Algorand implements a frugal oracle and its executions are strongly consistent with high probability while Byzcoin, PeerCensus, RedBelly and Hyperledger executions are strongly consistent.

2 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 [16]: 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. In this work we are interested in consistency criteria achievable in a distributed environment in which processes are sequential and communicate through message-passing.

2.1 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. The values that can be taken by the data type are encoded in the abstract state, taken in a set Z. 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, the corresponding transition in the transition system being formalized by a transition function τ. Second, operations can return values taken in an output alphabet B, which depend 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 (its side effect) and returns that element (its output).

The formal definition of abstract data types is as follows.

Definition 2.1. (Abstract Data Type T) An abstract data type is a 6-tuple $T = \langle A, B, Z, \xi_0, \tau, \delta \rangle$ where:

- A and B are countable sets called input alphabet and output alphabet;
- Z is a countable set of abstract states and ξ_0 is the initial abstract state;
- $\tau : Z \times A \to Z$ is the transition function;

- $\delta : Z \times A \to B$ is the output function.

Definition 2.2. (Operation) Let $T = \langle A, B, Z, \xi_0, \tau, \delta\rangle$ be an abstract data type. An *operation* of T is an element of $\Sigma = A \cup (A \times B)$. We refer to a couple $(\alpha, \beta) \in A \times B$ as α/β. We extend the transition function τ over the operations and apply τ on the operations input alphabet:

$$\tau_T : \begin{cases} Z \times \Sigma \to Z \\ (\xi, \alpha) \mapsto \tau(\xi, \alpha) \text{ if } \alpha \in A \\ (\xi, \alpha/\beta) \mapsto \tau(\xi, \alpha) \text{ if } \alpha/\beta \in A \times B \end{cases}$$

2.2 Sequential specification of an ADT

An abstract data type, by its transition system, defines the sequential specification of an object. 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. The language recognized by an ADT is the set of all possible words. This language defines the sequential specification of the ADT. More formally,

Definition 2.3. (Sequential specification $L(T)$) A finite or infinite sequence $\sigma = (\sigma_i)_{i\in D} \in \Sigma^\infty$, $D \in \mathbb{N}$ or $D \in \{0, \ldots, |\sigma| - 1\}$ is a *sequential history* of an abstract data type T if there exists a sequence of the same length $(\xi_{i+1})_{i\in D} \in Z^\infty$ (ξ_0 has already been defined has the initial state) of states of T such that, for any $i \in D$,

- the output alphabet of σ_i is compatible with ξ_i: $\xi_i \in \delta_T^{-1}(\sigma_i)$;
- the execution of the operation σ_i is such that the state changed from ξ_i to ξ_{i+1}: $\tau_T(\xi_i, \sigma_i) = \xi_{i+1}$.

The *sequential specification* of T is the set of all its possible sequential histories $L(T)$.

2.3 Concurrent histories of an ADT

Concurrent histories are defined considering asymmetric event structures, i.e., partial order relations among events executed by different processes [16].

Definition 2.4. (Concurrent history H) The execution of a program that uses an abstract data type T =⟨ A, B, Z, ξ_0, τ, δ⟩ defines a concurrent history $H = \langle \Sigma, E, \Lambda, \mapsto, \prec, \nearrow\rangle$, where

- $\Sigma = A \cup (A \times B)$ is a countable set of operations;
- E is a countable set of events that contains all the ADT operations invocations and all ADT operation response events;
- $\Lambda : E \to \Sigma$ is a function which associates events to the operations in Σ;
- $\mapsto$: is the process order relation over the events in E. Two events are ordered by $\mapsto$ if they are produced by the same process;

- $\prec$: is the operation order, irreflexive order over the events of E. For each couple $(e, e') \in E^2$, if e is an operation invocation and e' is the response for the same operation then $e \prec e'$, 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 $t < t'$ then $e \prec e'$;

- $\nearrow$: is the program order, irreflexive order over E, for each couple $(e, e') \in E^2$ with $e \neq e'$ if $e \mapsto e'$ or $e \prec e'$ then $e \nearrow e'$.

2.4 Consistency criterion

The 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. Specifically,

Definition 2.5. (Consistency criterion C) A consistency criterion is a function

$$C : \mathcal{T} \rightarrow \mathcal{P}(\mathcal{H})$$

where $\mathcal{T}$ is the set of abstract data types, $\mathcal{H}$ is a set of histories and $\mathcal{P}(\mathcal{H})$ is the sets of parts of $\mathcal{H}$.

Let $\mathcal{C}$ be the set of all the consistency criteria. An algorithm A_T implementing the ADT $T \in \mathcal{T}$ is C-consistent with respect to criterion $C \in \mathcal{C}$ if all the operations terminate and all the admissible executions are C-consistent, i.e. they belong to the set of histories $C(T)$.

3 BlockTree and Token oracle ADTs

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

3.1 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*. The height of a block refers to its distance to the root. We denote by b_k a block located at height k. By convention, the root of the BlockTree is denoted by b_0. Blocks are said valid if they satisfy a predicate P which is application dependent (for instance, in Bitcoin, a block is considered valid if it can be connected to the current blockchain and does not contain transactions that double spend a previous transaction). 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, $f \in \mathcal{F} : \mathcal{BT} \rightarrow \mathcal{BC}$; $f(bt)$ selects a blockchain bc from the BlockTree bt (note that b_0 is not returned). Selection function f is a parameter of the ADT which is encoded in the state and does not change over the computation. This reflects for instance the longest chain or the heaviest chain used in some blockchain implementations.

The following notations are also deeply 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;

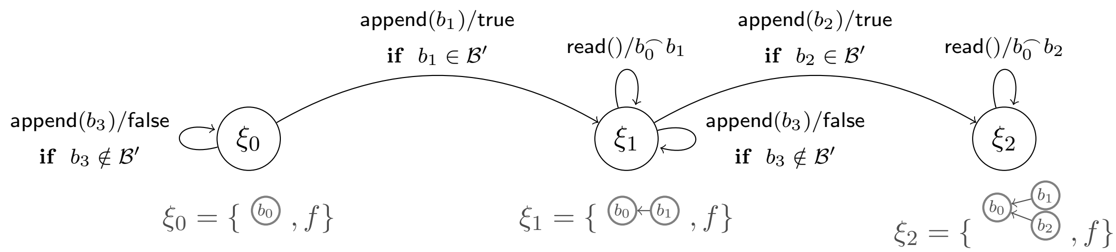


Figure 1: A possible path of the transition system defined by the Θ_F and Θ_P-ADTs. We use the following syntax on the edges: operation/output.

3.1.1 Sequential specification of the BlockTree

The sequential specification of the BlockTree is defined as follows.

Definition 3.1 (BlockTree ADT (*BT-ADT*))**.** The BlockTree Abstract Data Type is the 6-tuple BT-ADT=$\langle A = \{\mathsf{append}(b), \mathsf{read}()\colon b \in \mathcal{B}\}, B = \mathcal{BC} \cup \{\mathsf{true}, \mathsf{false}\}, Z = \mathcal{BT} \times \mathcal{F}, \xi_0 = (bt^0, f), \tau, \delta\rangle$, where the transition function $\tau : Z \times A \to Z$ is defined by

- $\tau((bt, f), \mathsf{append}(b)) = (\{b_0\}^\frown f(bt)^\frown \{b\}, f)$ if $b \in \mathcal{B}'$; (bt, f) otherwise;
- $\tau((bt, f), \mathsf{read}()) = (bt, f)$,

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

- $\delta((bt, f), \mathsf{append}(b)) = \mathsf{true}$ if $b \in \mathcal{B}'$; false otherwise;
- $\delta((bt, f), \mathsf{read}()) = \{b_0\}^\frown f(bt)$;
- $\delta((bt_0, f), \mathsf{read}()) = b_0$.

The semantic of the read and the append operations directly depend on the selection function $f \in \mathcal{F}$. In this work we let this function generic to suit the different blockchain implementations. In the same way, predicate P is let unspecified. The predicate P mainly abstracts the creation process of a block, which may fail or successfully terminate. This process will be further specified in Section 3.2.

3.1.2 Concurrent specification of a BT-ADT and consistency criteria

The concurrent specification of the BT-ADT is the set of concurrent histories. 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:

- $E(a^*, r^*)$ is an infinite set containing an infinite number of append() and read() invocation and response events;
- $E(a, r^*)$ is 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;

- $e_{inv}(o)$ and $e_{rsp}(o)$ indicate respectively the invocation and response event of an operation o; and $e_{rsp}(r) : bc$ denotes the returned blockchain bc associated with the response event $e_{rsp}(r)$;
- $\mathsf{score} : \mathcal{BC} \to \mathbb{N}$ denotes a monotonic increasing deterministic function that takes as input a blockchain bc and returns a natural number s as score of bc, which can be the height, the weight, etc. 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 that given two blockchains bc and bc' returns the score of the maximal common prefix between bc and bc';
- $bc \sqsubseteq bc'$ iff bc prefixes bc'.

BT Strong consistency. The BT Strong Consistency criterion is the conjunction of 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., satisfies predicate P) and has been inserted in the BlockTree with the $\mathsf{append}()$ operation. The Local monotonic read states that, given the sequence of $\mathsf{read}()$ operations at the same process, the score of the returned blockchain never decreases. The Strong prefix property states that for each couple of read operations, one of the returned blockchains is a prefix of the other returned one (i.e., the prefix never diverges). 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 $\mathsf{read}()$ operation r, the set of $\mathsf{read}()$ operations such that $e_{rsp}(r) \nearrow e_{inv}(r')$ that do not return blockchains with a score greater than s is finite. More formally, the BT Strong consistency criterion is defined as follows:

Definition 3.2 (BT Strong Consistency criterion (SC)). 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 following properties hold:

- **Block validity:** $\forall e_{rsp}(r) \in E, \forall b \in e_{rsp}(r) : bc, b \in \mathcal{B}' \wedge \exists e_{inv}(\mathsf{append}(b)) \in E$.
- **Local monotonic read:**
$$\forall e_{rsp}(r), e_{rsp}(r') \in E^2,\ \text{if } e_{rsp}(r) \mapsto e_{inv}(r'),\ \text{then } \mathsf{score}(e_{rsp}(r) : bc) \leq \mathsf{score}(e_{rsp}(r') : bc').$$
- **Strong prefix:**
$$\forall e_{rsp}(r), e_{rsp}(r') \in E^2, (e_{rsp}(r) : bc \sqsubseteq e_{rsp}(r) : bc') \vee (e_{rsp}(r) : bc \sqsubseteq e_{rsp}(r') : bc').$$
- **Ever growing tree:** $\forall e_{rsp}(r) \in E(a^*, r^*), s = \mathsf{score}(e_{rsp}(r) : bc)$ then
$$|\{e_{inv}(r') \in E \mid e_{rsp}(r) \nearrow e_{inv}(r'), \mathsf{score}(e_{rsp}(r') : bc) \leq s\}| < \infty.$$

Figure 2 shows a concurrent history H admissible by the BT Strong consistency criterion. In this example the score is the length l of the blockchain and the selection function f selects the longest blockchain, and in case of equality, selects the largest based on the lexicographical order. For ease of readability, we do not depict the $\mathsf{append}()$ operation. We assume the block validity property is satisfied. The Local monotonic read is easily verifiable as for each couple of read blockchains one prefixes the other. The first read() r operation, enclosed in a black rectangle, is taken as reference to check the consistency criterion (the criterion has to be iteratively verified for each read() operation). Let l be the score of the blockchain returned by r. We can identify two sets, enclosed in rectangles defined by different patterns: *(i)* the finite sets of read() operations such that the score associated to each blockchain returned is smaller than or equal to l, and *(i)* the infinite set of read() operations such that the score is greater than l. We can iterate the same reasoning for each read() operation in H. Thus H satisfies the Ever growing tree property.

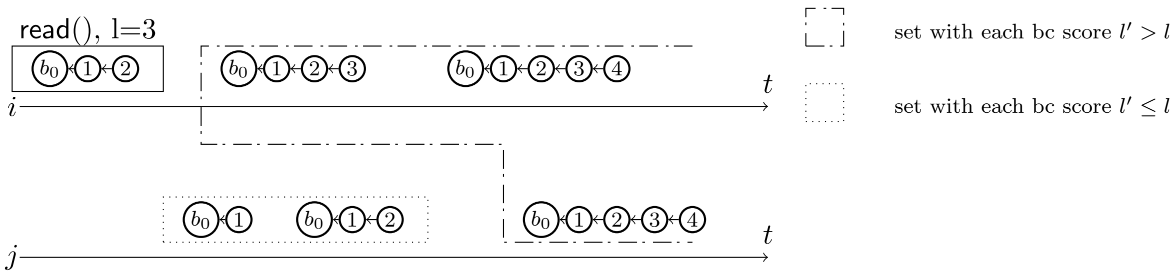


Figure 2: Concurrent history that satisfies the BT Strong consistency criterion. In such scenario f selects the longest blockchain and the blockchain score is length l.

BT Eventual consistency. The BT Eventual consistency criterion is the conjunction of the block validity, the Local monotonic read and the Ever growing tree of the BT Strong consistency criterion together with the Eventual prefix which states 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 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.

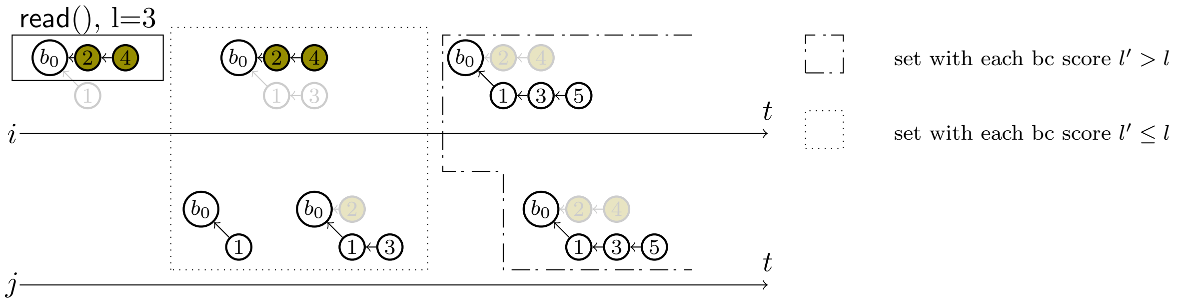
Definition 3.3 (Eventual prefix property)**.** Given a concurrent history $H = \langle \Sigma, E(a, r^*), \Lambda, \mapsto, \prec, \nearrow \rangle$ of the system that uses a BT-ADT, we denote by s, for any read operation $r \in \Sigma$ such that $\exists e \in E(a, r^*), \Lambda(r) = e$, the score of 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,

$$|\{(e_{rsp}(r_h), e_{rsp}(r_k)) \in E_r^2 | h \neq k, \mathsf{mpcs}(e_{rsp}(r_h) : bc_h, e_{rsp}(r_k) : bc_k) < s\}| < \infty$$

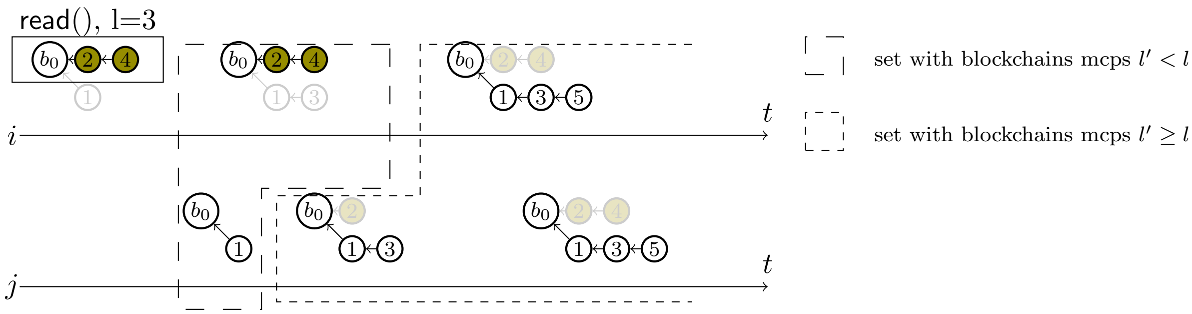
The Eventual prefix properties captures the fact that two or more concurrent blockchains can co-exist in a finite interval of time, but that ly all the participants adopts a same branch for each cut of the history. This cut of the history is defined by a read that picks up a blockchain with a given score.

Based on this definition, the BT Eventual consistency criterion is defined as follows:

Definition 3.4 (BT Eventual consistency criterion $\lozenge C$)**.** 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.



(a) Sets for the Ever Growing Tree property.



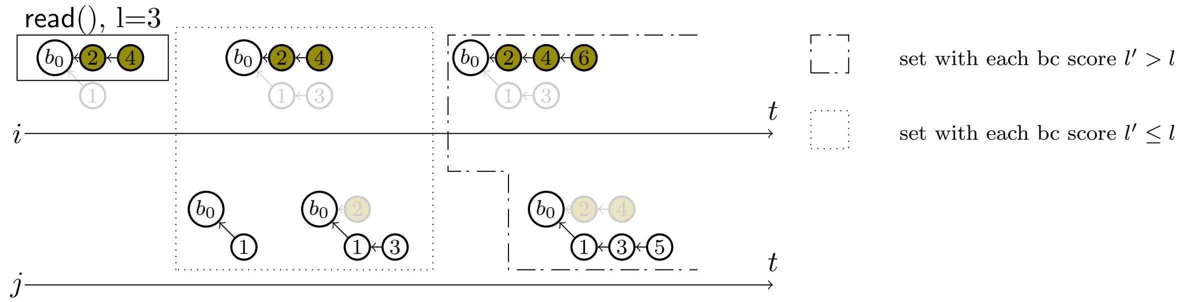
(b) Sets for the Eventual Prefix Property.

Figure 3: Concurrent history that satisfies the Eventual BT consistency criterion. In such scenario f selects the longest blockchain and the blockchain score is the length l. In case (a) and case (b) the concurrent history is the same but different sets are outlined.

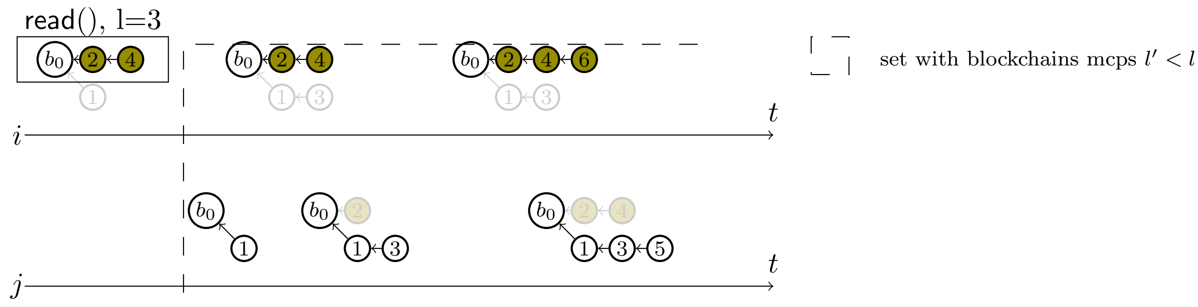
Figure 3 shows a concurrent history that satisfies the Eventual prefix property but not the Strong prefix one. Strong Prefix is not satisfied as blockchain[1] $b_0^\frown 1$ returned from the first read() at process j is not a prefix of blockchain $b_0^\frown 2^\frown 4$ returned from the first read at process i. Note that we adopt the same conventions as for the example depicted in Figure 2 regarding the score, length and append() operations. We assume that the Block validity property is satisfied. The Local monotonic read property is easily verifiable. In both Figures 3a and 3b, the first read() r operation at i, enclosed in a black rectangle, is taken as reference to check the consistency criterion (the criterion has to be iteratively verified for each read() operation). Let l be the score of the blockchain returned by r. In Figure 3b we can identify two sets, enclosed in rectangles defined by different patterns: *(i)* the finite set of read() operations sharing a maximum common prefix score (mcps) smaller than l (the set to check for the satisfiability of the Eventual Prefix property), and *(ii)* the infinite set of read() operations such that for each couple of them bc, bc', $\mathsf{mcps}(bc, bc') \geq l$. We can iterate the same reasoning for each read() operation in H. Thus H satisfies the Eventual Prefix property. Figure 4 shows a history that does not satisfy any consistency criteria defined so far.

Relationships between $\Diamond C$ and SC. Let us denote by $\mathcal{H}_{\Diamond C}$ and by $\mathcal{H}_{SC}$ the set of histories satisfying respectively the $\Diamond C$ and the SC consistency criteria.

[1]For ease of readability we extend the notation $b_i^\frown b_j$ to represent concatenated blocks in a blockchain.



(a) Sets for the Ever Growing Tree property.



(b) Sets for the Eventual Prefix Property.

Figure 4: Concurrent history that does not satisfy any BT consistency criteria. In such scenario f selects the longest blockchain and the blockchain score is the length l.

Theorem 3.1. Any history H satisfying SC criterion satisfies $\Diamond C$ and $\exists H$ satisfying $\Diamond C$ that does not satisfy SC, i.e., $\mathcal{H}_{SC} \subset \mathcal{H}_{\Diamond C}$.

Proof $\Diamond C \leq SC$ implies that $\mathcal{H}_{SC} \subset \mathcal{H}_{\Diamond C}$, and $\mathcal{H}_{SC} \subset \mathcal{H}_{\Diamond C}$ implies that $\forall H \in \mathcal{H}_{SC} \Rightarrow H \in \mathcal{H}_{\Diamond C}$. By hypothesis, H verifies the Ever Growing Tree property, thus $\forall e_{rsp}(r) \in E(a^*, r^*)$ with $s = score(e_{rsp}(r) : bc)$ then set $\{e_{inv}(r') \in E | e_{rsp}(r) \nearrow e_{inv}(r'), score(e_{rsp}(r'') : bc) \leq s\}$ is finite, and thus, there is an infinite set $\{e_{inv}(r') \in E | e_{rsp}(r) \nearrow e_{inv}(r'), score(e_{rsp}(r'') : bc) > s\}$. The Strong prefix property guarantees that $\forall e_{rsp}(r), e_{rsp}(r') \in H, (e_{rsp}(r) : bc \sqsubseteq e_{rsp}(r) : bc') \vee (e_{rsp}(r) : bc \sqsubseteq e_{rsp}(r') : bc')$, thus in this infinite set, all the $\mathsf{read}()$ operations return blockchains sharing the same maximum prefix whose score is at least $s+1$, which satisfies the Eventual prefix property. The Eventual Prefix property demands that for each $\forall e_{rsp}(r) \in E(a, r^*)$ with $s = score(e_{rsp}(r) : bc)$ there is an infinite set defined as $\{(e_{rsp}(r_h), e_{rsp}(r_k)) \in E_r^2 | h \neq k, \mathsf{mpcs}(e_{rsp}(r_h) : bc_h, e_{rsp}(r_k) : bc_k) \geq s\}$ where E_r denotes the set of response events of read operations that occurred after r response. To conclude the proof we need to find a $H \in \mathcal{H}_{\Diamond C}$ and $H \notin \mathcal{H}_{SC}$. Any H in which at least two $\mathsf{read}()$ operations return a blockchain sharing the same prefix but diverging in their suffix violate the Strong prefix property, which concludes the proof. $\square_{Theorem\ 3.1}$

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. Moreover, an append is successful only if the input block is valid with respect to a predicate. This means that histories with no append operations are trivially admitted. In the following we will introduce a new abstract data type called Token Oracle that when combined with the BlockTree will help in *(i)* validating blocks and *(ii)* controlling forks. We will first formally introduce the Token Oracle in Section 3.2 and then we will define the properties on the BlockTree augmented with the Token Oracle in Section 3.4.

3.2 Token oracle Θ-ADT

In this section we formalize the Token Oracle Θ to capture the creation of blocks in the BlockTree structure. The block creation process requires that the 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 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 as general as possible, we model blocks as objects. More formally, when a process wants to access a generic object obj_h, it invokes the $\mathsf{getToken}(obj_h, obj_\ell)$ operation with object obj_ℓ from set $\mathcal{O} = \{obj_1, obj_2, \dots\}$. If $\mathsf{getToken}(obj_h, obj_\ell)$ operation is successful, it returns an object $obj_\ell^{tkn_h} \in \mathcal{O}'$, where *(i)* is the token required to access object obj_h and *(ii)* each object $obj_k \in \mathcal{O}'$ is valid with respect to predicate P, i.e. $P(obj_k) = \top$. We say that a token is *generated* each time it is provided to a process and it is *consumed* when the oracle grants the right to connect it to the previous object. Each token can be consumed at most once. To consume a token we define the token consumption $\mathsf{consumeToken}(obj_\ell^{tkn_h})$ operation, where the consumed token tkn_h is the token required for the object obj_h. A maximal number of tokens k for an object obj_h is managed by the oracle. The $\mathsf{consumeToken}(obj_\ell^{tkn_h})$ side-effect on the state is the decrement of k by one for object obj_h.

In the following 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, assures controls that no more than k token can be consumed for each object.

Θ_P when combined with the BlockTree abstract data type will only help in validating blocks, while Θ_F manages tokens in a more controlled way to guarantee that no more than k forks can occur on a given block.

3.2.1 Θ_P-ADT and Θ_F-ADT definitions

For both oracles, when $\mathsf{getToken}(obj_k, obj_h)$ operation is invoked, the oracle provides a token with a certain probability p_{α_i} where α_i is a "merit" parameter characterizing the invoking process i. [2] 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. [3] When a $\mathsf{getToken}(obj_k, obj_h)$ 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 token is provided to the process if that cell contains tkn.

Both oracles also enjoy an infinite array of counters, one for each object, which is decreased each time a token is consumed for a specific object. When the counter reaches 0 then no more

[2] The merit parameter can reflect for instance the hashing rate of the invoking process.

[3] We assume a pseudo-random 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.

tokens can be consumed for that object. For a sake a generality, Θ_P is defined as Θ_F with $k = \infty$ while for Θ_F each counter is initialized to $k \in \mathbb{N}$.

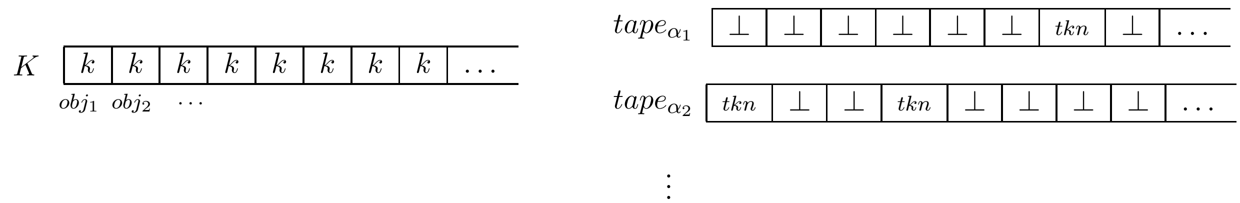


Figure 5: The Θ_F abstract state. The infinite K array, where at the beginning each value is initialized to k and the infinite set of infinite tapes, one for each merit α_i in $\mathcal{A}$.

We first introduce some definitions and notations.

- $\mathcal{O} = \{obj_1, obj_2, \dots\}$, infinite set of generic objects uniquely identified by their index i;
- $\mathcal{O}' \subset \mathcal{O}$, the subset of objects valid with respect to predicate P, i.e. $\forall obj'_i \in \mathcal{O}', P(obj'_i) = \top$.
- $\mathfrak{T} = \{tkn_1, tkn_2, \dots\}$ infinite set of tokens;
- $\mathcal{A} = \{\alpha_1, \alpha_2, \dots\}$ an infinite set of rational values;
- $\mathcal{M}$ is a countable not empty set of mapping functions $m(\alpha_i)$ that generate an infinite pseudo random tape $tape_{\alpha_i}$ such that the probability to have in a cell the string tkn is related to a specific α_i, $m \in \mathcal{M} : \mathcal{A} \to \{tkn, \perp\}^*$;
- $K[\ ]$ is a infinite array of counters (one per object). All the counters are initialized with a $k \in \mathbb{N}$, where k is a parameter of the oracle ADT;
- $pop : \{tkn, \perp\}^* \to \{tkn, \perp\}^*$, $pop(a \cdot w) = w$;
- $head : \{tkn, \perp\}^* \to \{tkn, \perp\}^*$, $head(a \cdot w) = a$;
- $dec : \{K\} \times \mathbb{N} \to \{K\}$, $dec(K, i) = K : K[i] = K[i] - 1$ **if** $K[i] > 0$; $K[i] = 0$ otherwise;
- $get : \{K\} \times \mathbb{N} \to \mathbb{N}$, $get(K, i) = K[i]$;

Definition 3.5. (Θ_F-ADT Definition). The Θ_F Abstract Data type is the 6-tuple Θ_F-ADT $=$(A$=$ $\{\mathsf{getToken}(obj_h, obj_\ell), \mathsf{consumeToken}(obj_\ell^{tkn_h}) : obj_h, obj_\ell^{tkn_h} \in \mathcal{O}', obj_\ell \in \mathcal{O}, tkn_h \in \mathfrak{T}\}$, B$=$ $\mathcal{O}' \cup Boolean$, Z$=$ $m(A)^* \times \{K\} \cup \{pop, head, dec, get\}$, ξ_0, τ, δ), where the transition function $\tau : Z \times A \to Z$ is defined by

- $\tau((\{tape_{\alpha_1}, \dots, tape_{\alpha_i}, \dots\}, K), \mathsf{getToken}(obj_h, obj_\ell)) = (\{tape_{\alpha_1}, \dots, pop(tape_{\alpha_i}), \dots\}, K)$ with α_i the merit of the invoking process;
- $\tau((\{tape_{\alpha_1}, \dots, tape_{\alpha_i}, \dots\}, K), \mathsf{consumeToken}(obj_\ell^{tkn_h})) = (\{tape_{\alpha_1}, \dots, tape_{\alpha_i}, \dots\}, dec(K, h))$, if $tkn_h \in \mathfrak{T}$; $\{(\{tape_{\alpha_1}, \dots, tape_{\alpha_i}, \dots\}, K)\}$ otherwise.

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

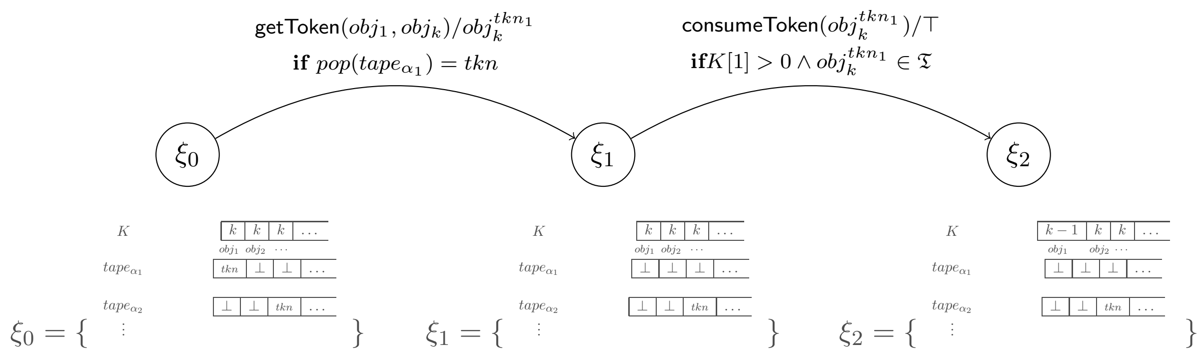


Figure 6: A possible path of the transition system defined by the Θ_F and Θ_P-ADTs. We use the following syntax on the edges: operation/output.

```
                          upon append(b_ℓ):
                              ...
Init:                         while (token = ⊥):
    token ← ⊥;                    token ← getToken(b_h ← last_block(f(bt)), b_ℓ);
    bt_i ← b_0;               consumeToken(token) ∧ {b_0}⌢f(bt)⌢{b};
    ...                       token ← ⊥;
                              ...
```

Figure 7: A generic implementation fragment of the BT-ADT that employs a Θ oracle to implement the append() operation.

- $\delta((\{tape_{\alpha_1}, \ldots, tape_{\alpha_i}, \ldots\}, K), \mathsf{getToken}(obj_h, obj_\ell)) = obj_\ell^{tkn_h} : obj_\ell^{tkn_h} \in \mathcal{O}', tkn_h \in \mathfrak{T}$, if $head(tape_{\alpha_i}) = tkn$ with α_i the merit of the invoking process; $\perp$ otherwise;
- $\delta((\{tape_{\alpha_1}, \ldots, tape_{\alpha_i}, \ldots\}, K), \mathsf{consumeToken}(obj_\ell^{tkn_h})) = \top$ if $tkn_h \in \mathfrak{T}$ and $get(K, h) > 0$; $\perp$ otherwise.

Definition 3.6. (Θ_P-ADT Definition). The Θ_P Abstract Data type is defined as the Θ_F-ADT with $k = \infty$.

Figure 6 shows a possible path of the transition system defined by the Θ_F and Θ_P-ADTs.

3.3 BT-ADT augmented with Θ Oracles

In this section 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. We are considering that a generic implementation (cf. Figure 7) of the BT-ADT invokes the getToken$(b_k \leftarrow$last_block$(f(bt)), b_\ell)$ operation as long as it returns a token on b_k, i.e., ${b_\ell}^{tkn_h}$ which is valid block in $\mathcal{B}'$. Once obtained, the token is consumed and the append terminates, i.e. the block ${b_\ell}^{tkn_h}$ is appended to the blockchain $f(bt)$. Notice that those two operations and the concatenation occur atomically.
We say that the *BT-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.

Definition 3.7 ($\mathfrak{R}(BT\text{-}ADT, \Theta_F)$ refinement)**.** Given the BT-ADT=$\langle A, B, Z, \xi_0, \tau, \delta\rangle$, and the Θ_F-ADT =$(A^\Theta, B^\Theta, Z^\Theta,\ \xi_0^\Theta, \tau^\Theta, \delta^\Theta)$, we have $\mathfrak{R}(BT - ADT, \Theta_F)$=$\langle A' = A \cup A^\Theta, B' = B \cup B^\Theta, Z' = Z \cup Z^\Theta,\ \xi_0' = \xi_0 \cup \xi_0^\Theta, \tau', \delta'\rangle$, where the transition function $\tau' : Z' \times A' \to Z'$ is defined by

- $\tau_a = \tau'((\{tape_{\alpha_1}, \dots, tape_{\alpha_i}, \dots\}, K, bt, f), \mathsf{getToken}(b_k \leftarrow \text{last\_block}(bt), b_\ell)) = (\{tape_{\alpha_1}, \dots, pop(tape_{\alpha_i}), \dots\}, K, bt, f)$;
- $\tau_b = \tau'((\{tape_{\alpha_1}, \dots, tape_{\alpha_i}, \dots\}, K, bt, f), \mathsf{consumeToken}(b_\ell^{tkn_h})) =$
  $(\{tape_{\alpha_1}, \dots, tape_{\alpha_i}, \dots\}, dec(K, h), \{b_0\}^\frown f(bt)^\frown\{b\}, f)$ if $tkn_h \in \mathfrak{T}$;
  $(\{tape_{\alpha_1}, \dots, tape_{\alpha_i}, \dots\}, K, bt, f)$ otherwise;
- $\tau'((\{tape_{\alpha_1}, \dots, tape_{\alpha_i}, \dots\}, K, bt, f), \mathsf{append}(b)) = \tau_b \circ \tau_a^*$
  where $\tau_b \circ \tau_a^*$ is the repeated application of τ_a until $\delta_a((\{tape_{\alpha_1}, \dots, tape_{\alpha_i}, \dots\}, K, bt, f), \mathsf{getToken}(b_k \leftarrow \text{last\_block}(bt), b_\ell)) = b_\ell^{tkn_h}$ concatenated with the τ_b application;
- $\tau'(\{tape_{\alpha_1}, \dots, tape_{\alpha_i}, \dots\}, K, bt, f, \mathsf{read}()) = bt$.

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

- $\delta_a = \delta'((\{tape_{\alpha_1}, \dots, tape_{\alpha_i}, \dots\}, K, bt, f), \mathsf{getToken}(b_k \leftarrow \text{last\_block}(bt), b_\ell)) = b_\ell^{tkn_h} : b_\ell^{tkn_h} \in \mathcal{B}', tkn_h \in \mathfrak{T}$, if $head(tape_{\alpha_i}) = tkn$ with α_i the merit of the invoking process; $\perp$ otherwise;
- $\delta_b = \delta'((\{tape_{\alpha_1}, \dots, tape_{\alpha_i}, \dots\}, K, bt, f), \mathsf{consumeToken}(obj_\ell^{tkn_h})) = \top$ if $tkn_h \in \mathfrak{T}$ and $get(K, h) > 0$; $\perp$ otherwise;
- $\delta'((\{tape_{\alpha_1}, \dots, tape_{\alpha_i}, \dots\}, K, bt, f), \mathsf{append}(b)) = \delta_b \circ \delta_a^*$, where $\delta_b \circ \delta_a^*$ is the repeated application of δ_a until $\delta_a((\{tape_{\alpha_1}, \dots, tape_{\alpha_i}, \dots\}, K, bt, f), \mathsf{getToken}(\text{last\_block}(bt), b)) = b_\ell^{tkn_h}$ concatenated with the δ_b application;
- $\delta'((\{tape_{\alpha_1}, \dots, tape_{\alpha_i}, \dots\}, K, bt, f), \mathsf{read}()) = \{b_0\}^\frown f(bt)$;
- $\delta'((\{tape_{\alpha_1}, \dots, tape_{\alpha_i}, \dots\}, K, bt_0, f), \mathsf{read}()) = b_0$.

Definition 3.8 ($\mathfrak{R}(BT\text{-}ADT, \Theta_P)$ refinement)**.** Same definition as the $\mathfrak{R}(BT\text{-}ADT, \Theta_F)$ refinement.

Theorem 3.2 (Append termination)**.** Let $H = \langle \Sigma, E, \Lambda, \mapsto, \prec, \nearrow\rangle$ be any concurrent history generated by $\mathfrak{R}(BT\text{-}ADT, \Theta_P)$. Then each invoked $\mathsf{append}()$ operation in H eventually successfully terminates.

Proof We prove the theorem by considering the defined refinement (cf. Figure 7) where *(i)* there are a infinite number of $\mathsf{getToken}()$ invocations for object obj and *(ii)* given a valid block as input parameter, the $\mathsf{consumeToken}()$ operation always successfully terminates by definition of Oracle Θ_P (as k is infinite). From the properties of the pseudo random sequences of tapes, if there are an infinite number of $\mathsf{getToken}()$ invocations for object obj then there exists at least one response for which $\mathsf{getToken}()$ operation returns a valid block, thus when passed as input of the $\mathsf{consumeToken}()$ operation, such operation successfully terminates. $\Box_{Theorem\ 3.2}$

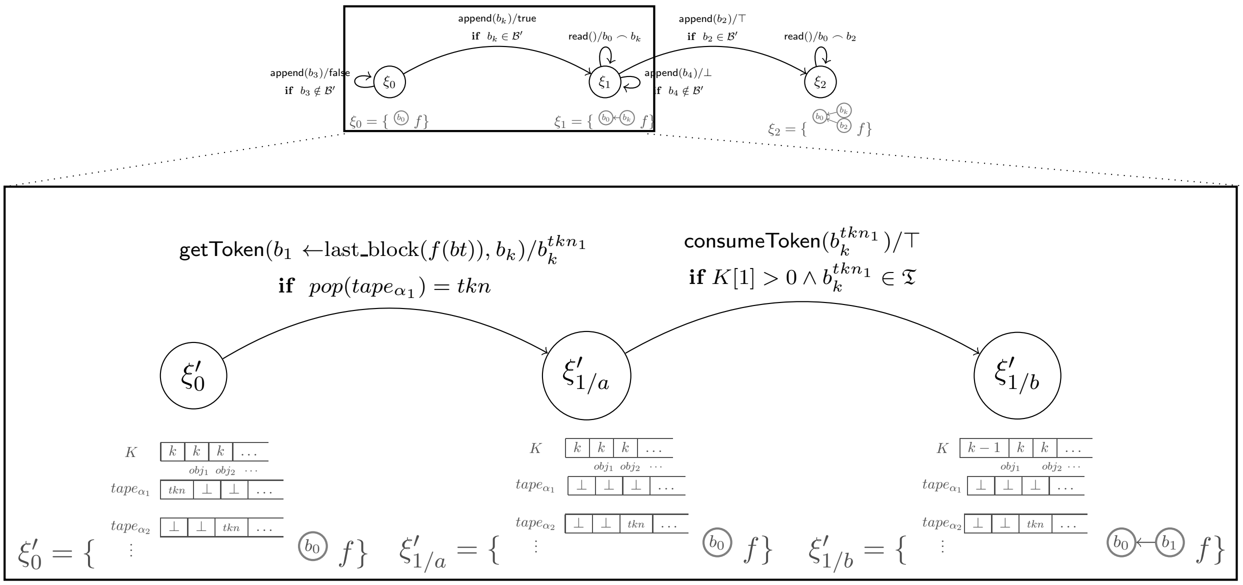


Figure 8: Refinement of the append() operation. We use the following syntax on the edges: operation/output.

Definition 3.9 (k-Fork Coherence)**.** A concurrent history $H = \langle \Sigma, E, \Lambda, \mapsto, \prec, \nearrow \rangle$ of the BT-ADT composed with Θ_F-ADT satisfies the *k-Fork Coherence* if there are at most k append() operations that return $\top$ for the same token.

Theorem 3.3 (k-Fork Coherence)**.** Each concurrent history $H = \langle \Sigma, E, \Lambda, \mapsto, \prec, \nearrow \rangle$ of the BT-ADT composed with a Θ_F-ADT satisfies the *k-Fork Coherence.*

Proof We prove the theorem by considering the defined refinement (cf. Figure 7) where *(i)* there are a infinite number of getToken() invocations for object *obj* and *(ii)* given a valid block as input parameter, the consumeToken() operation successfully terminates if it has been invoked less than k times for the same token. From the properties of the pseudo random sequences of tapes, if there are an infinite number of getToken() invocations for object *obj* then there exists at least one response for which getToken() operation returns a token t, which, when passed as input of the consumeToken() operation it successfully terminates if at most $k-1$ tokens t have been already consumed. $\square_{Theorem\ 3.3}$

3.4 Hierarchy

In this section we define a hierarchy between different BT-ADT satisfying different consistency criteria when augmented with different oracle ADT. We use the following notation: BT-ADT$_{SC}$ and BT-ADT$_{\Diamond C}$ to refer respectively to BT-ADT generating concurrent histories that satisfies the SC and the $\Diamond C$ consistency criteria. When augmented with the oracles we have the following four typologies, where for the *frugal* oracle we explicit the value of k: $\mathfrak{R}(\text{BT-ADT}_{SC}, \Theta_{F,k})$, $\mathfrak{R}(\text{BT-ADT}_{SC}, \Theta_P)$, $\mathfrak{R}(\text{BT-ADT}_{\Diamond C}, \Theta_P)$, $\mathfrak{R}(\text{BT-ADT}_{\Diamond C}, \Theta_{F,k})$.

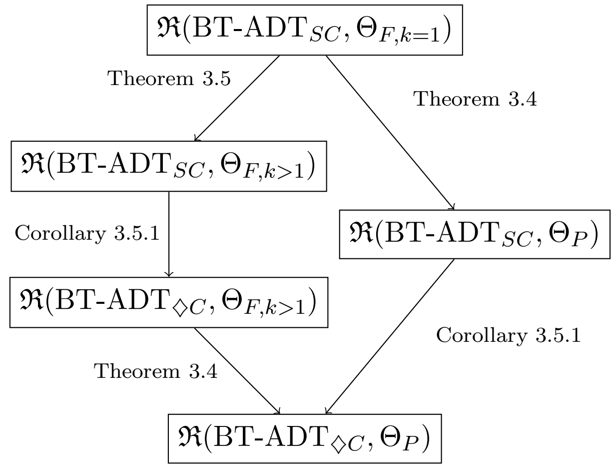


Figure 9: $\mathfrak{R}(\text{BT-ADT}, \Theta)$ Hierarchy.

In the following we want study the relationship among the different refinements. Without loss of generality, let us consider only the set of histories $\hat{\mathcal{H}}^{\mathfrak{R}(\text{BT-ADT},\Theta)}$ such that each history $\hat{H}^{\mathfrak{R}(\text{BT-ADT},\Theta)} \in \hat{\mathcal{H}}^{\mathfrak{R}(BT-ADT,\Theta)}$ is purged from the unsuccessful $\mathsf{append}()$ response events (i.e., such that the returned value is $\perp$). Let $\hat{\mathcal{H}}^{\mathfrak{R}(\text{BT-ADT},\Theta_F)}$ be the concurrent set of histories generated by a BT-ADT enriched with Θ_F-ADT and let $\hat{\mathcal{H}}^{\mathfrak{R}(\text{BT-ADT},\Theta_P)}$ be the concurrent set of histories generated by a BT-ADT enriched with Θ_P-ADT.

Theorem 3.4. $\hat{\mathcal{H}}^{\mathfrak{R}(\text{BT-ADT},\Theta_F)} \subseteq \hat{\mathcal{H}}^{\mathfrak{R}(\text{BT-ADT},\Theta_P)}$.

Proof The proof follows from Theorem 3.2 and 3.3 considering that $\mathfrak{R}(BT, \Theta_F)$ generates histories with an infinite number of $\mathsf{append}()$ operations that successfully terminate while $\mathfrak{R}(BT, \Theta_F)$ generates history with at most k $\mathsf{append}()$ operations that successfully terminate. $\Box_{Theorem\ 3.4}$

Theorem 3.5. If $k_1 \leq k_2$ then $\hat{\mathcal{H}}^{\mathfrak{R}(\text{BT-ADT},\Theta_{F,k_1})} \subseteq \hat{\mathcal{H}}^{\mathfrak{R}(\text{BT-ADT},\Theta_{F,k_2})}$.

Proof The proof follows from Theorem 3.3 applying the same reasoning as for the proof of Theorem 3.4 with $k_1 \leq k_2$. $\Box_{Theorem\ 3.5}$

Finally, from Theorem 3.1 the next corollary follows.

Corollary 3.5.1. $\hat{\mathcal{H}}^{(\mathfrak{R}(\text{BT-ADT}_{SC},\Theta)} \subseteq \hat{\mathcal{H}}^{\mathfrak{R}(\text{BT-ADT}_{\Diamond C},\Theta)}$.

Combining Theorem 3.1 and Theorem 3.4 we obtain the hierarchy depicted in Figure 9, where the gray combinations are removed thanks to Theorem 4.5.

4 Implementing BT-ADTs

4.1 Blockchain system model

We consider a message-passing system composed of an arbitrary large but finite set of n processes, $\Pi = \{p_1, \ldots, p_n\}$. The passage of time is measured by a fictional global clock (*e.g.*, that spans the set of natural integers). Processes in the system do not have access to the fictional global time. Each process of the distributed system executes a single instance of a distributed protocol $\mathcal{P}$ composed of a set of algorithms, i.e., each process is running an algorithm. Processes can exhibit a Byzantine behavior (i.e., they can arbitrarily deviate from the protocol $\mathcal{P}$ they are suppose to run). A process affected by a Byzantine behavior is said to be faulty, otherwise we refer to such process as non-faulty or correct. We make no assumption on the number of failures that can occur during the system execution. Processes communicate by exchanging messages.

The BlockTree being now a shared object replicated at each process, we note by bt_i 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 updates. We also use subscript i to indicate that the operation occurred at process i: $\mathsf{update}_i(b_g, b_i)$ 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_i generated on a process p_i, sent through $\mathsf{send}_i(b_g, b_i)$, and received through a $\mathsf{receive}_j(b_g, b_i)$, takes effect on the local replica bt_j of p_j with the operation $\mathsf{update}_j(b_g, b_i)$.

We assume a generic implementation of the update operation: when process i locally updates its BlockTree bt_i with the valid block b_i (returned from the consumeToken() operation), we write $\mathsf{update}_i(b, b'_i)$. When a process j execute the $\mathsf{receive}_j(b, b_i)$ operation, it locally updates its BlockTree bt_j by invoking the $\mathsf{update}_j(b, b_i)$ operation.

In the remaining part of the work we consider implementations of BT-ADT in a Byzantine failure model where the set of events is restricted as follows.

Definition 4.1. The execution of the system that uses the BT-ADT =(A, B, Z, ξ_0, τ, δ) in a Byzantine failure model defines the concurrent history $H = \langle \Sigma, E, \Lambda, \mapsto, \prec, \nearrow \rangle$ (see Definition 2.4) where we restrict E 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 the predicate P and, *(iv) send*, *receive* and *update* events generated at correct processes.

4.2 Communication Abstractions

We now define the properties that each history H generated by a BT-ADT satisfying the Eventual Prefix Property has to satisfy and then we prove their necessity.

Definition 4.2 (Update Agreement)**.** A concurrent history $H = \langle \Sigma, E, \Lambda, \mapsto, \prec, \nearrow \rangle$ of the system that uses a BT-ADT satisfies the Update Agreement if satisfies the following properties:

R1. $\forall \mathsf{update}_i(b_g, b_i) \in H, \exists \mathsf{send}_i(b_g, b_i) \in H$;

R2. $\forall \mathsf{update}_i(b_g, b_j) \in H, \exists \mathsf{receive}_i(b_g, b_j) \in H$ such that $\mathsf{receive}_i(b_g, b_j) \mapsto \mathsf{update}_i(b_g, b_j)$;

R3. $\forall \mathsf{update}_i(b_g, b_j) \in H$, $\exists \mathsf{receive}_k(b_g, b_j) \in H, \forall k$.

In the following, for ease of notation we consider that the selection function $f \in \mathcal{F}$ returns directly also the genesis block.

Lemma 4.1. Property R1 or Property R2 are necessary conditions for any protocol $\mathcal{P}$ to implement a BT-ADT generating histories H satisfying the Eventual Prefix property.

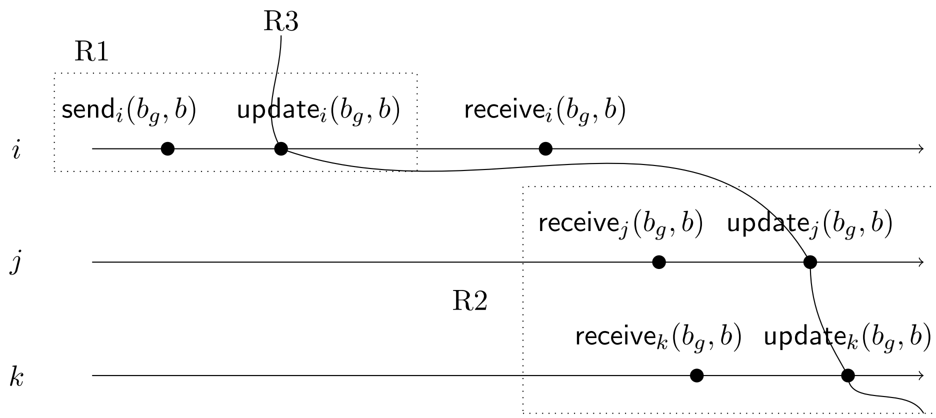


Figure 10: Example of concurrent history that satisfies R1,R2 and R3, the Update Agreement properties.

Proof Let us assume that there exists a protocol $\mathcal{P}$ implementing a BT-ADT that generates histories H satisfying Eventual Prefix property but not Property R1 or Property R2. Thus, in H there is some update u that is not sent to the other processes (R1) or once received, u is not locally applied (R2). Let us consider the following history where R1 is not verified and process i issues the first update event in H.
Let us construct the following execution history H. i issues the $\mathsf{update}_i(b_0, b'_i)$ (thus $bt_i = b_0 \frown b'_i$) but not the $\mathsf{send}_i(b_0, b'_i)$ event. It follows that if there is no $\mathsf{send}_i(b_0, b'_i)$ event in H then in H are no present any $\mathsf{receive}_j(b_0, b)$ events, $j \neq i$ and thus not process $j \neq$ can issue $\mathsf{update}_j(b_0, b'_i)$ (on the other side, if R2 is not satisfied, even if the the $\mathsf{receive}_j(b_0, b)$ event occur then $\mathsf{update}_j(b_0, b'_i)$ may not occur), thus $\forall j \neq i, bt_j = b_0$. Let us assume that i preforms a $\mathsf{read}()$ operation, the selection function $f \in \mathcal{F}$ is applied on $bt_i = b_0 \frown b'_i$. By the score function definition it follows that $f(b_0 \frown b'_i) > f(b_0)$. Thus if i issues a $\mathsf{read}()$ operation after $\mathsf{update}_i(b_0, b'_i)$ it returns a blockchain such that $\mathsf{score}(b_0 \frown b)$ and the possible infinite $\mathsf{read}()$ operations issued by other processes always return blockchain such that $\mathsf{score}(b_0)$, violating the Eventual Prefix property. The construction of H can be completed iterating the same reasoning for an infinite number of $\mathsf{append}()$ operation issued by i, thus H violates the Eventual Prefix Property leading to a contradiction. $\square_{Lemma\ 4.1}$

Lemma 4.2. Property R3 is a necessary condition for any protocol $\mathcal{P}$ to implement a BT-ADT generating histories H satisfying the Eventual Prefix property.

Proof Let us assume that there exists a protocol $\mathcal{P}$ implementing a BT-ADT that generates histories H satisfying Eventual Prefix property but not Property R3. Thus, in H there is some $\mathsf{update}_i(b, b'_i)$ u at some process i such that the $\mathsf{receive}_j(b, b'_i)$ events do not occur at all processes $j \neq i$.
Let us consider a system composed by three processes, i, j and k. The system execution generates the following history H where R3 is not verified. In particular, in H are present the $\mathsf{update}_i(b_0, b'_i)$, $\mathsf{receive}_j(b_0, b'_i)$ events but there is no any $\mathsf{receive}_k(b_0, b'_i)$ event. It follows that $bt_i = bt_j = b_0 \frown b'_i$ and $bt_z = b_0$. We apply the same argument as for Lemma 4.1. Let us assume that j and k perform $\mathsf{read}()$ operations. Such operation returns the result of $f(bt_j)$ and $f(bt_k)$ respectively. By the score function definition it follows that $\mathsf{score}(b_0 \frown b'_i) > \mathsf{score}(b_0)$. If j issues a $\mathsf{read}()$ operation after

$\mathsf{update}_j(b_0, b'_i)$ it returns a blockchain with $\mathsf{score}(b_0^\frown b)$ and the other $\mathsf{read}()$ operations issued by k will always return blockchain with $\mathsf{score}(b_0)$. The construction of H can be completed iterating the same reasoning for an infinite number of $\mathsf{append}()$ operation issued by i, thus H violates the Eventual Prefix Property leading to a contradiction. $\Box_{Lemma\ 4.2}$

Theorem 4.3. 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 proof follows directly from Lemma 4.1, Lemma 4.2 and the definition of Eventual BT consistency criterion. $\Box_{Theorem\ 4.3}$

Considering Theorem 4.3 and Theorem 3.1 the next Corollary follows.

Corollary 4.3.1. There not exists a concurrent history $H = \langle \Sigma, E, \Lambda, \mapsto, \prec, \nearrow \rangle$ of the system that uses a BT-ADT that satisfies the Strong BT consistency criterion but not the Update Agreement.

In the following we consider a communication primitive that is inspired by the Liveness properties of the reliable broadcast [3]. We will prove that this abstraction is necessary to implement Eventual BT Consistency.

Definition 4.3 (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): $\forall \mathsf{receive}_i(b, b_j) \in H, \forall k \exists \mathsf{receive}_k(b, b_i) \in H$

In other words, if a correct process i sends a message m then i eventually receives m and if a message m is received by some correct process, them m is eventually received by every correct process.

Theorem 4.4. The LRC abstraction is necessary to for any BT-ADT implementation that generates concurrent histories that satisfies the BT Eventual Consistency criterion.

Proof The proof done by generating a concurrent history H that violates the LCR properties and showing that H also violate the Update Agreement properties. For Theorem 4.3 the Update Agreement properties are necessary condition to implement BT-ADT that generates concurrent histories that satisfies the BT Eventual Consistency criterion.
Let us consider H where at process n occurs the event $\mathsf{update}_n(b, b_n)$ and $\mathsf{send}_n(b, b_n)$ and where the LRC2 property is not satisfied. If LRC2 is violated then in H we can have that there exist some process i at which occurs the $\mathsf{receive}_i(b, b_n)$ event and some process j at which never occurs the $\mathsf{receive}_j(b, b_n)$ event. Since at process n occurred the event $\mathsf{update}_n(b, b_n)$, then for the R3 property then for each process k $\mathsf{update}_n(b, b_n)$ has to occur. For R2 the $\mathsf{update}_m(b, b_n)$ event at some process m has to be preceded by a $\mathsf{receive}_m(b, b_n)$ event at the same process m. Since by hypothesis not at all processes m the $\mathsf{receive}_m(b, b_n)$ occurs then the property is violated, violating the Update Agreement properties, which are necessary conditions to implement BT-ADT that generates concurrent histories that satisfies the BT Eventual Consistency criterion, which concludes the proof.

$\Box_{Theorem\ 4.4}$

Finally, from Theorem 3.1 and Theorem 4.4 the next Corollary follows.

Corollary 4.4.1. The LRC abstraction is necessary to for any BT-ADT implementation that generates concurrent histories that satisfies the BT Strong Consistency criterion.

4.3 System model and hierarchy

Theorem 4.5. There does no exist an implementation of a BT-ADT that generates histories satisfying the BT Strong consistency if forks occur.

Proof Let us assume that there exist a BT-ADT implementation that satisfies the BT Strong consistency criterion despite the occurrence of forks. Let us now construct the following history H generated by the system execution at two correct processes i and j. At the beginning $bt_i = bt_j = b_0$. At the same time instant t_0 both processes invoke $\mathsf{append}(b_1)$ and $\mathsf{append}(b_2)$ operations respectively and $b_1, b_2 \in \mathcal{B}'$. By definition, the $\mathsf{append}()$ operation applies a selection function $f \in \mathcal{F}$ to select the block from the BlockTree to which the new block has to be appended, in this case such block is $f(bt_i) = f(bt_j) = f(b_0) = b_0$. By construction, $b_i, b_j \in \mathcal{B}'$, let us assume that a fork occurs and both $\mathsf{append}()$ operations take place and update events are triggered. From Theorem 4.3, each update has to be sent to the other processes. Since there are no synchrony assumptions on the communication channels then the reception of messages at different processes can occur at different time instants. Let us consider that H contains the following ordered events: $update_i(b_0, b_j) \mapsto update_i(b_0, b_i)$ and $update_j(b_0, b_i) \mapsto update_j(b_0, b_j)$. It follows that at a time instant t it can occur that $bt_i = b_0^\frown b_j$ and $bt_j = b_0^\frown b_i$. Let us finally assume that at time t both i and j issue a $\mathsf{read}()$ operation. By definition it returns the result of the selection function f to the BlockTree. For both processes the BlockTree is a blockchain, thus the $\mathsf{read}()$ operations returns $b_0^\frown b_j$ at i and $b_0^\frown b_i$ at j violating the Strong Prefix property leading to a contradiction. Thus, there no exists an implementation of a BT-ADT that generates histories satisfying the BT Strong consistency if forks occur. $\square_{Theorem\ 4.5}$

Thanks to Theorem 4.5 we can eliminate from the hierarchy in Figure 9 both $\mathfrak{R}(\text{BT-ADT}_{SC}, \Theta_P)$ and $\mathfrak{R}(\text{BT-ADT}_{SC}, \Theta_{F,k>1})$, since in both cases the Θ-ADT employed allows forks, thus such enriched ADTs can not generate histories that satisfies the BT Strong consistency criterion. The resulting hierarchy is depicted in Figure 11.

5 Mapping with existing Blockchain-like systems

This section completes this work by illustrating the mapping between different existing systems and the specifications and abstractions presented in this paper. The following table summarizes the mapping between different existing systems and these abstractions. More details are given in the following sections. In those sections we refer to a permissionless system as a system where the cardinality of the process set is not a-priori known and each process can read and append into the blockchain. When we do not consider permissionless systems we explicitly state the differences.

5.1 Bitcoin

Bitcoin [14] is the pioneer of blockchain systems. Any 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 (implemented with TCP), which models a partially synchronous setting [7].

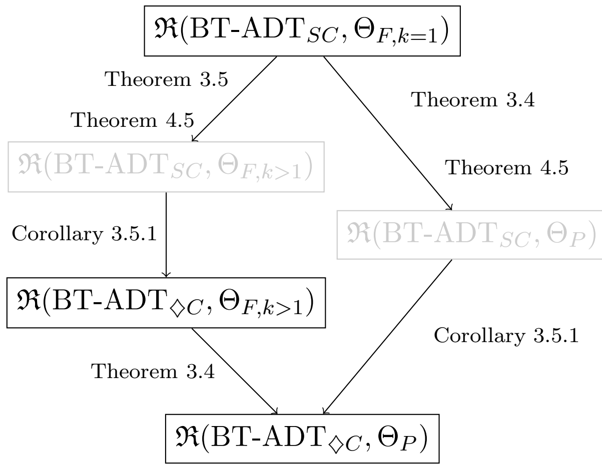


Figure 11: $\mathfrak{R}(\text{BT-ADT}, \Theta)$ Hierarchy. In gray the combinations impossible in a message-passing system

Table 1: Mapping of existing systems. Each of these systems assumes at least a light reliable communication.

References	Refinement
Bitcoin [14]	$\mathfrak{R}(BT\text{-}ADT_{\Diamond C}, \Theta_P)$
Ethereum [19]	$\mathfrak{R}(BT\text{-}ADT_{\Diamond C}, \Theta_P)$
Algorand [9]	$\mathfrak{R}(BT\text{-}ADT_{SC}, \Theta_{F,k=1})$ SC with v.h.p
ByzCoin [13]	$\mathfrak{R}(BT\text{-}ADT_{SC}, \Theta_{F,k=1})$
PeerCensus [6]	$\mathfrak{R}(BT\text{-}ADT_{SC}, \Theta_{F,k=1})$
Redbelly [5]	$\mathfrak{R}(BT\text{-}ADT_{SC}, \Theta_{F,k=1})$
Hyperledger [2]	$\mathfrak{R}(BT\text{-}ADT_{SC}, \Theta_{F,k=1})$

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 f selects returns the blockchain which has required the most computational work, guaranteeing that concurrenrt blocks can only refer to the most recently appended blocks of the blockchain returned by a read() operation. Garay and al [8] have shown, under a synchronous environment assumption, that Bitcoin ensures Eventual consistency criteria. The same conclusion applies as well for the FruitChain protocol [15], which proposes a protocol similar to BitCoin except for the rewarding mechanism.

5.2 Ethereum

Ethereum [19] is a permissionless blockchain. Processes are characterized by their merit parameter represented by α_p (once normalized as $\sum_{p \in V} \alpha_p = 1$). Contrarily to Bitcoin, where this merit parameter is representative of a computational power, that is this ability to quickly compute hash functions, in Ethereum this ability is bounded by this ability to move data in memory. This proof-of-work mechanism is especially designed for commodity hardware. Any process $p \in V$ is allowed to read the BlockTree and append blocks to the BlockTree. Processes communicate through reliable FIFO authenticated channels (implemented with TCP), which models a partially synchronous setting [7]. 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 Ethereum implements a Prodigal Oracle. The f selects returns the blockchain which has required the most work (see Section 10 of [19]), guaranteeing that concurrenrt blocks can only refer to the most recently appended blocks of the blockchain returned by a read() operation. This function is implemented through GHOST algorithm [18]. Kiayias has shown [12], under a synchronous environment assumption, that GHOST protocol enjoys both common prefix and chain growth properties. Ethereum thus ensures the Eventual consistency criteria.

5.3 ByzCoin

ByzCoin [13] is a permissionless blockchain. Processes are characterized by their computational power represented by α_p (once normalized as $\sum_{p \in V} \alpha_p = 1$). Byzcoin assumes a semi synchronous environment, that is, in every period of length b there must be a strongly synchronous period of length $s < b$. The block creation process is separated from the transaction validation one. The former one is realized by a proof-of-work mechanism (similar to the Bitcoin's one), and the latter one is achieved by a Byzantine tolerant algorithm (i.e., a variant of PBFT [4]) which creates micro blocks made of transactions.

The getToken operation is implemented by a proof-of-work mechanism. Due to the PoW mechanism, several key blocks can be concurrently created. The consumeToken operation guarantees that during the synchronous periods of the semi-synchronous setting (those synchronous periods ensure that everyone receives all the concurrent key blocks in a short period of time), a single key block will be appended to the BlockTree by relying on a deterministic function f which selects the key block whose digest (fingerprint) has the smallest least significant bits among the concurrent key blocks. Under those assumptions, Byzcoin is an implementation of a strongly consistent BlockTree composed with a Frugal Oracle, with $k = 1$.

Note that transactions do not belong to key blocks but to microblocks which are created by a variant of PBFT where *(i)* the committee members are the miners of the last w appended key blocks in the BlockTree as returned by a read() operation; *(ii)* each committee member receives a voting share for each block it has created blocks among these w ones, and *(iii)* committee members are organized on a tree rooted at the leader, and *(iv)* this leader is the process that invoked the last successful consumeToken operation.

5.4 Algorand

Algorand [9] is an algorithm dedicated to permisionless blockchains. Users are characterized by the quantity of coins (stake) they own, represented by α_p once normalized as $\sum_{p \in V} \alpha_p = 1$. Algorand assumes a synchronous setting (rounds) in order to ensure that *(i)* with overwhelming probability all users agree on the same transactions (safety property) and *(ii)* new transactions are added to the blockchain (liveness property). Note that safety holds even in a semi synchronous environment. Users communicate among themselves through reliable communication channels (implemented via TCP). Algorand algorithm relies on two main ingredients: a cryptographic sortition and a variant of a byzantine agreement algorithm. The cryptographic sortition implements the getToken operation by selecting the block proposer. This is achieved by selecting at random a committee (that is a small fraction of users weighed by their currency balance α_p, which boils down to a proof-of-stake mechanism) and providing them a random priority, so that with high probability, the highest priority committee member will be in charge of proposing the new block for the current round. The variant of Byzantine agreement algorithm BA* implements the consumeToken operation, that is the commitment to append this new valid block in the blockchain. BA* guarantees that in a favorable environment (strongly synchronous environment augmented with synchronized clocks), if all honest participants have received the same valid block, then this block will be appended to the blockchain (see Lemma 2 [10]). On the other hand, if there is no agreement on that block (because the highest priority committee member is malicious or the network is not strongly synchronous), then BA* may create forks with probability less than 10^{-7} (Theorem 2 [10]). This makes Algorand a probabilistic implementation of a strongly consistent BlockTree composed with a Frugal Oracle, with $k = 1$.

5.5 PeerCensus

PeerCensus [6] is a permissionless blockchain. Processes are characterized by their computational power represented by α_p (once normalized as $\sum_{p\in V} \alpha_p = 1$). PeerCensus assumes a semi synchronous environment, that is, in every period of length b there must be a strongly synchronous period of length $s < b$. PeerCensus is not strictly speaking a blockchain-based algorithm (as Bitcoin or Byzcoin), in the sense that it does not store a sequence of application transactions, but provides a secure and fully distributed timestamping service. This service is implemented by a dynamic Byzantine tolerant consensus algorithm which tracks the committee members of the consensus algorithm through the creation of chained key blocks. The getToken operation is implemented by a proof-of-work mechanism, and the consumeToken operation, implemented by the Byzantine consensus, commits a single key block among the concurrent ones, that is returns true for a single token, as long as no more than a 1/3 of the committees members are Byzantine (secure state). Theorem 1 [6] states that the secure state is reachable with high probability if the computational power owned by the adversary, α_A, is less than 1/3. Thus under these assumptions PeerCensus implements a strongly consistent BlockTree composed with a Frugal Oracle, with $k = 1$. Note however that in [1] the authors have analyzed the probability that PeerCensus reaches a secure state by examing the composition of successive quorums, and have shown that this probability is decreasing as a function of α_A. For instance, if $\alpha_A = 1/4$, then the probability that PeerCensus reaches a secure state is only equal to 1/3.

5.6 Red Belly

Red Belly [5] 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 $p \in M$ as a merit parameter set to $\alpha_p = 1/|M|$ while each process $p \in V \setminus M$ has a merit parameter $\alpha_p = 0$. Processes are asynchronous (i.e., there is no assumption on their respective computational speed) and are connected with partially synchronous [7] (i.e., messages are delivered in unknown but finite time), reliable and authenticated communication channels. 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 [5]).

5.7 HyperLedger Fabric

HyperLedger Fabric [2] is a system allowing to deploy and operate persmissioned blockchains. Any process $p \in V$ is allowed to read the BlockTree, however, only a subset of $M \subseteq V$ is allowed to append blocks to the BlockTree. Every process of M has the same merit parameter $\alpha_M = 1/|M|$ while processes of $V \setminus M$ have a null merit parameter. HyperLedger Fabric assumes eventual synchrony and reliable channels. Transactions are executed by a dedicated set of processes called endorsers. Executed transactions are then ordered through atomic broadcast primitive so as to gather them into a block. HyperLedger Fabric relies on a leader election to determine which process will generate the next block. Transactions are appended in a block until a stop condition is met. A stop condition refers either on a maximal number of transactions in a block or a maximal elapsed time since the first transaction included in the block. The block is then broadcasted and a new block is created to gather new incomping transactions. By construction, HyperLedger Fabric ensures that a unique token ($k = 1$) is consumed, thus HyperLedger Fabric implement a strongly consistent BlockTree.

6 Conclusions

The paper presented an extended formal specification of blockchains and derived interesting conclusion on their implementability in message passing distributed systems. We hope that this work will be a reference for future implementations of blockchains, helping as well their formal validation.

References

[1] E. Anceaume, T. Lajoie-Mazenc, R. Ludinard, and B. Sericola. Safety Analysis of Bitcoin Improvement Proposals. In *Proceedings of the 15th IEEE International Symposium on Network Computing and Applications (NCA)*, 2016.

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

[3] Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011.

[4] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance and Proactive Recovery. *Journal ACM Transactions on Computer Systems (TOCS)*, 2002.

[5] T. Crain, V. Gramoli, M. Larrea, and M. Raynal. (Leader/Randomization/Signature)-free Byzantine Consensus for Consortium Blockchains. `http://csrg.redbellyblockchain.io/doc/ConsensusRedBellyBlockchain.pdf`, 2017.

[6] C. Decker, J. Seidel, and R. Wattenhofer. Bitcoin Meets Strong Consistency. In *Proceedings of the 17th International Conference on Distributed Computing and Networking Conference (ICDCN)*, 2016.

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

[8] J. A. Garay, A. Kiayias, and N. Leonardos. The Bitcoin Backbone Protocol: Analysis and Applications. In *Proceedings of the 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2015.

[9] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017.

[10] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. Technical report, MIT CSAIL, 2017. `https://people.csail.mit.edu/nickolai/papers/gilad-algorand-eprint.pdf`.

[11] A. Girault, G. Gößler, R. Guerraoui, J. Hamza, and D-A. Seredinschi. Why You Can't Beat Blockchains: Consistency and High Availability in Distributed Systems. `http://arxiv.org/abs/1710.09209`.

[12] A. Kiayias and G. Panagiotakos. On Trees, Chains and Fast Transactions in the Blockchain. `http://eprint.iacr.org/2016/545`, 2016.

[13] E. Kokoris-Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford. Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing. In *Proceedings of the 25th USENIX Security Symposium*, 2016.

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

[15] Rafael Pass and Elaine Shi. Fruitchains: A fair blockchain. In *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, pages 315–324, 2017.

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

[17] M. Perrin, A. Mostefaoui, and C. Jard. Causal Consistency: Beyond Memory. In *21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2016.

[18] Y. Sompolinsky and A. Zohar. Accelerating Bitcoin's Transaction Processing. Fast Money Grows on Trees, Not Chains. `https://eprint.iacr.org/eprint-bin/getfile.pl?entry=2013/881&version=20140101:161740&file=881.pdf`, 2013.

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