On Finality in Blockchains

Emmanuelle Anceaume, Antonella Del Pozzo, Thibault Rieutord, Sara Tucci-Piergiovanni

On Finality in Blockchains

Emmanuelle Anceaume[1], Antonella Del Pozzo[2],
Thibault Rieutord[2], Sara Tucci-Piergiovanni[2]

[1]CNRS, Univ Rennes, Inria, IRISA, Rennes, France
[2]Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France

Abstract

There exist many forms of Blockchain finality conditions, from deterministic to probabilistic terminations. To favor availability against consistency in the face of partitions, most blockchains only offer probabilistic eventual finality: blocks may be revoked after being appended to the blockchain, yet with decreasing probability as they sink deeper into the chain. Other blockchains favor consistency by leveraging the immediate finality of Consensus – a block appended is never revoked – at the cost of additional synchronization.

In this paper, we focus on necessary and sufficient conditions to implement a blockchain with deterministic eventual finality, which ensures that selected main chains at different processes share a common increasing prefix. This is a much weaker form of finality that allows us to provide a solution in an asynchronous system subject to unlimited number of byzantine failures. We study stronger forms of eventual finality as well and show that it is unfortunately impossible to provide a *bounded displacement*. By bounded displacement we mean that the (unknown) number of blocks that can be revoked from the current blockchain is bounded. This problem reduces to consensus or eventual consensus depending on whether the bound is known or not. We also show that the classical selection mechanism, such as in Bitcoin, that appends blocks at the longest chain is not compliant with a solution to eventual finality.

1 Introduction

This paper focuses on blockchain finality, which refers to the time when it becomes impossible to remove a block that has previously been appended to the blockchain. Blockchain finality can be deterministic or probabilistic, immediate or eventual. Informally, immediate finality guarantees, as its name suggests, that when a block is appended to a local copy, it is immediately finalized and thus will never be revoked in the future. Most of the permissionned blockchains satisfy the deterministic form of immediate consistency, including Red Belly blockchain [8] and Hyperledger Fabric blockchain [3], while the probabilistic form of immediate consistency is typically achieved by permissionless pure proof-of-stake blockchains such as Algorand [7]. Designing blockchains with immediate finality favors consistency against availability in presence of transient partitions of the system. It leverages the properties of Consensus (i.e a decision value is unique and agreed by everyone), at the cost of synchronization constraints. On the other hand eventual finality ensures that all local copies of the blockchain share a common increasing prefix, and thus finality of

their blocks increases as more blocks are appended to the blockchain. The majority of cryptoassets blockchains, with Bitcoin [15] and Ethereum [19] as celebrated examples, guarantee eventual finality with some probability: blocks may be revoked after being appended to the blockchain, yet with decreasing probability as they sink deeper into the chain.

Formalization of blockchains in the lens of distributed computing has been recognized as an extremely important topic [12]. Garay et al. [10] have been the first to analyse the Bitcoin backbone protocol and to define invariants this protocol has to satisfy to verify with high probability an eventual consistent prefix, i.e. probabilistic eventual finality. The authors analyzed the protocol in a synchronous system, while Pass et al. [17] extended this line of work considering a more adversarial network. Anta et al. [4] proposed a formalization of distributed ledgers modeled as an ordered list of records along with implementations for sequential consistency and linearizability using a total order broadcast abstraction. Not related to the blockchain data structure, authors of [11] formalized the notion of cryptocurrency showing that Consensus is not needed.

While probabilistic eventual finality has been widely studied in the context of Bitcoin [10, 5, 17], some few studies have started to lay the foundations of the computation power of blockchains with deterministic eventual finality consistency. Anceaume et al. [2] have been the first to propose a formal blockchain specification as a composition of abstract data types all together with a hierarchy of consistency criteria. This work captured the convergence process of two distinct classes of blockchain systems: the class providing strong prefix (for each pair of chains returned at two different processes, one is the prefix of the other) and the class providing eventual prefix, in which the common prefix eventually converges. Interestingly, the authors of [2] show that to solve strong prefix, Consensus is needed, however the work does not address the solvability of eventual prefix, which is the focus of this paper.

The objective of this paper is thus to push further this line of inquiry by presenting an in-depth study of deterministic eventual finality. We reinvestigate in Section 2 the definition of (deterministic) eventual prefix consistency presented in [2] to fit both the context in which an infinite number of blocks are appended to the blockchain. We introduce the notion of bounded displacement, which informally says that the number of blocks that can be revoked from the current blockchain is bounded. Providing solutions that guarantee a known bound of the displacement reveals to be an important crux in the construction of blockchains. Specifically, in Section 3 we show that known bounded displacement eventual prefix consistency, that is eventual prefix consistency guaranteeing a bounded and known displacement, is equivalent to Consensus. We also demonstrate that unknown bounded displacement eventual prefix consistency, that is eventual prefix consistency guaranteeing a bounded but unknown displacement, is equivalent to eventual strong prefix consistency. Finally we show that eventual strong prefix consistency is stronger than Eventual Consensus, an abstraction that captures eventual agreement among all participants. Then in Section 4 we provide an algorithm that guarantees eventual prefix consistency in an asynchronous environment with an unbounded number of Byzantine processes. We are not aware of any such solution in the literature. We also show that it is impossible to build a blockchain that guarantees eventual prefix consistency based on the longest chain rule. Note that such a selection function is used by many cryptoassets

blockchains. Finally, we discuss impossibilities and possibilities of unbounded displacement eventual prefix consistency. In particular, we propose an algorithm that solves unknown bounded displacement eventual prefix in an eventually synchronous environment in presence of less than a majority of Byzantine processes. Once again, we are not aware of any such solution in the literature. To summarize, we close the gap between eventual prefix and strong prefix consistencies.

2 Definitions

2.1 Preliminary Definitions

Similarly to [2], we describe a blockchain object as an abstract data type which allows us to completely characterize a blockchain by the operations it exports [14]. The basic idea underlying the use of abstract data types is to specify shared objects using two complementary facets: 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 [16]. Prior to presenting the blockchain abstract data type we first recall the formalization used to describe an abstract data type (ADT).

Abstract data types. An abstract data type (ADT) is a tuple of the form $T = (A, B, Z, z_0, \tau, \delta)$. Here A and B are countable sets called the *inputs* and *outputs*. Z is a countable set of abstract object *states*, $z_0 \in Z$ being the initial state of the object. The map $\tau : Z \times A \to Z$ is the *transition function*, specifying the effect of an input on the object state and the map $\delta : Z \times A \to B$ is the *output function*, specifying the output returned for a given input and object local state. The input represents an operation with its parameters, where (i) the operation can have a side-effect that changes the abstract state according to transition function τ and (ii) the operation can return values taken in the output B, which depends on the state in which it is called and the output function δ.

Concurrent histories of an ADT. Concurrent histories are defined considering asymmetric event structures, i.e., partial order relations among events executed by different processes.

Definition 1. (Concurrent history H) *The execution of a program that uses an abstract data type $T = \langle A, B, Z, \xi_0, \tau, \delta \rangle$ 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, irreflexive order over the events of E. Two events $(e, e') \in E^2$ are ordered by $\mapsto$ if they are produced by the same process, $e \neq e'$ and e happens before e', that is denoted as $e \mapsto e'$.*

- $\prec$: *is the operation order, irreflexive order over the events of* E. *For each couple* $(e, e') \in E^2$ *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* $e \nearrow e'$.

2.2 The blockchain ADT

Following [2], we represent a blockchain structure as a a tree of blocks. Indeed, while consensus-based blockchains prevent forks or branching in the tree of blocks, most blockchain systems based on proof-of-work allow the occurrence of forks to happen hence presenting blocks under a tree structure. The blockchain object is thus defined as a blocktree abstract data type (Blocktree ADT). However, we modify the original version of the blocktree ADT proposed in [2] so that distinct selection functions $f_r()$ and $f_a()$ for respectively the read() and append() operations exist. As will be shown latter in the paper, by relying on different functions to select the blockchain to be returned by a read() operation and to select the chain of the blocktree to which the new block is appended, this allows us to present the first algorithm that constructs a blockchain satisfying deterministic eventual finality in an asynchronous system in presence of an unbounded number of Byzantine processes.

2.2.1 Sequential Specification

A blocktree data structure is a directed rooted tree $bt = (V_{bt}, E_{bt})$ where V_{bt} represents a set of blocks and E_{bt} a set of edges such that each block has a single path towards the root of the tree b_0 called the genesis block. Let $\mathcal{B}$ be the countable and non empty set of uniquely identified blocks and let $\mathcal{BC}$ be the 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. The structure is equipped with two operations append() and read(). Operation append(b) adds the block $b \notin bt$ to V_{bt} and adds the edge (b, b') to E_{bt} where $b' \in V_{bt}$ is returned by the append selection function $f_a()$ applied to bt. Operation read() returns the chain bc selected by the read selection function $f_r()$ applied to bt. In the following the chain bc returned by a read() operation r is often denoted by r/bc. Only blocks satisfying some validity predicate P can be appended to the tree. The definition of the Blocktree (BT) ADT amended with the two types of selection functions is as follows.

Definition 2. *(Blocktree ADT (BT-ADT) The Blocktree Abstract Data Type is the* 6*-tuple* $\text{BT} - \text{ADT} = \{A = \{append(b), read()/bc \in \mathcal{B}\}, B = \mathcal{BC} \cup \{\top, \perp\}, Z = \mathcal{BT}, \xi_0 = b_0, \tau, \delta\}$, *where the transition function* $\tau : Z \times A \to Z$ *is defined by*

$$\tau(bt, read()) = bt$$

$$\tau(bt, append(b)) = \begin{cases} (V_{bt} \cup \{b\}, E_{bt} \cup \{b, last\_block(f_a(bt))\}) \text{ if } P(f_a(bt)^\frown b) \\ bt \text{ otherwise,} \end{cases}$$

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

$$\delta(bt, read()) = f_r(bt)$$
$$\delta(bt, append(b)) = \begin{cases} \top \text{ if } P(f_a(bt)^\frown b) \\ \bot \text{ otherwise.} \end{cases}$$

The read selection $f_r()$ takes as argument the blocktree and returns a chain, that is a sequence of blocks starting from the genesis block to a leaf block of the blocktree. The chain returned by the read() operation is called the blockchain. The append selection function $f_a()$ takes as argument the blocktree and returns a chain of blocks. Function *last_block*() takes as argument a chain of blocks and returns the last appended block of the chain.

Predicate P is an application-dependent predicate used to verify the validity of the chain obtained by appending the new block b to the chain returned by $f_a()$. In Bitcoin for instance this predicate embeds the logic to verify that the obtained chain does not contain double spending or overspending transactions.

Note that we do not need to add the validity check during the read operation in the sequential specification, because in absence of concurrency the validity check during the append operation is enough.

2.2.2 Concurrent specification and consistency criteria

The concurrent specification of a blocktree abstract data type is the set of concurrent histories. A blocktree consistency criterion is a function that returns the set of concurrent histories admissible for a blocktree abstract data type. We define three consistency criteria for the blocktree: *BT Eventual Prefix consistency*, *BT Strong Prefix consistency* and *BT Eventual Strong 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;
- o_{inv} and o_{rsp} indicate respectively the invocation and response event of an operation o; and in particular for the read() operation, r_{rsp}/bc denotes the returned blockchain bc associated with the response event r_{rsp} and for the append() operation $a_{inv}((b))$ denotes the invocation of the append operation having b as input parameter;
- length : $\mathcal{BC} \to \mathbb{N}$ denotes a monotonic increasing deterministic function that takes as input a blockchain bc and returns a natural number as length of bc. Increasing monotonicity means that length$(bc^\frown\{b\}) >$ length(bc);
- $bc \sqsubseteq bc'$ iff bc prefixes bc'.
- $bc[i]$ refers to the $i-th$ block in the blockchain bc.

Eventual Prefix (EP) Consistency

Definition 3 (BT Eventual Prefix Consistency (EP) criterion)**.** *A concurrent history* $H = \langle \Sigma, E, \Lambda, \mapsto, \prec, \nearrow \rangle$*, of a system that uses a BT-ADT, verifies the BT Eventual Prefix Consistency criterion if the following properties hold:*

- **Chain validity:**

  $\forall r_{rsp} \in E, P(r_{rsp}/bc)$.

  Each returned chain is valid.

- **Chain integrity:**

  $\forall r_{rsp} \in E, \forall b \in r_{rsp}/bc : b \neq b_0, \exists a_{inv}(b) \in E, a_{inv}(b) \nearrow r_{rsp}$.

  If a block different from the genesis block is returned, then an append *operation has been invoked with this block as parameter. This property is to avoid the situation in which* reads *return blocks never* appended.

- **Eventual prefix:**

  $\forall E \in E(a, r^*) \cup E(a^*, r^*), \forall r_{rsp}/bc, \forall i \in \mathbb{N} : bc[i] \neq \bot, \exists r'_{rsp}, \forall r''_{rsp} : r'_{rsp} \nearrow r''_{rsp}, ((r'_{rsp}/bc)[i] = (r''_{rsp}/bc)[i])$.

  In all the histories in which the number of read *invocations is infinite, then for any non empty read chain position i, there exists a* read r'/bc' *from which all the subsequent* reads r''/bc'' *will return the same block at position i, i.e.* $bc'[i] = bc''[i]$.

- **Ever growing tree:**

  $\forall E \in E(a^*, r^*), \forall k \in \mathbb{N}, \exists r \in E : \mathbf{length}(r_{rsp}/bc) > k$.

  In all the histories in which the number of append *and* read *invocations is infinite, for each length k, there exists a* read *that returns a chain with length greater than k. This property avoids the trivial scenario in which the length of the chain remains unchanged despite the occurrence of an infinite number of* append *operations. This can happen for instance if the tree is built as a star with infinite branches of bounded length.*

Strong Prefix (SP) Consistency

Definition 4 (BT Strong Prefix Consistency criterion (SP))**.** *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 Chain validity, Chain integrity, Ever growing tree (as defined for EP) and the following property hold:*

- **Strong prefix:**

  $\forall r_{rsp}, r'_{rsp} \in E^2, (r'_{rsp}/bc' \sqsubseteq r_{rsp}/bc) \vee (r_{rsp}/bc \sqsubseteq r'_{rsp}/bc')$.

  For each pair of returned blockchains, one blockchain is the prefix of the other.

Eventual Strong Prefix (ESP) Consistency

Definition 5 (BT Eventual Strong Prefix Consistency criterion (ESP))**.** *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 Chain validity, Chain integrity, Ever growing tree (as defined for EP) and the following property hold:*

- **Eventual strong prefix:**

  $\forall E \in E(a, r^*) \cup E(a^*, r^*)$, $\exists r_{rsp} \in E, \forall r'_{rsp}, r''_{rsp} \in E^2 : r_{rsp} \nearrow r'_{rsp} \wedge r_{rsp} \nearrow r''_{rsp}, (r''_{rsp}/bc' \sqsubseteq r'_{rsp}/bc) \vee (r'_{rsp}/bc \sqsubseteq r''_{rsp}/bc')$.

  In all histories with an infinite number of reads*, there exists a* read *r from which for each pair of returned blockchains, one blockchain is the prefix of the other.*

Bounded displacement

Informally, the bounded displacement says that for any two reads r/bc and r'/bc' such that r precedes r', then by pruning the last *dis* blocks from bc the obtained chain is a prefix of bc'. Note that constant *dis* can be initially known or not.

Definition 6. *Bounded displacement*

- $\exists \mathsf{dis} \in \mathbb{N}, \forall r_{rsp}, r'_{rsp} \in E : r_{rsp} \nearrow r'_{rsp}, \forall i \in \mathbb{N} : i \leq \mathbf{length}(r_{rsp}/bc) - \mathsf{dis}, (r_{rsp}/bc)[i] = (r'_{rsp}/bc')[i]$.

We will show in the following that satisfying the eventual prefix consistency criterion plus the bounded displacement property with known *dis* boils down to getting the strong prefix consistency criterion. This is because any algorithm implementing the read selection function f_r, can safely select the current chain but the last *dis* blocks. Eventual prefix consistency criterion plus the known bounded displacement is designated in the sequel by *known bounded displacement eventual prefix consistency.*

The eventual prefix consistency criterion plus the bounded displacement property with unknown *dis* boils down to get the eventual strong prefix consistency criterion. In this case a simple algorithm implementing the read operation always returns half of the selected longest chain. Since chains are always growing, the number of removed blocks increases up to reaching *dis*. In the sequel, eventual prefix consistency criterion plus the unknown bounded displacement is designated by *unknown bounded displacement eventual prefix consistency.*

In the following section we prove the above-mentioned equivalences more formally and study relationships to known problems such as Consensus and Eventual Consensus to determine the assumptions on the system needed to implement blocktree consistency models.

3 (Eventual) Consensus Reductions

In this Section we investigate the impact of the bounded displacement property combined with the eventual prefix consistency problems. In particular, we derive

that when the displacement is known, such problem is equivalent to Consensus, while when unknown, this problem is stronger than Eventual Consensus [9].

3.1 Known Bounded Displacement and Consensus

Theorem 1. *Known bounded displacement eventual prefix is equivalent to Consensus.*

Proof. We first show how to solve strong prefix given a solution $\mathcal{P}_{EP}$ for known bounded displacement eventual prefix and then the reciprocal direction. Indeed, the equivalence between the strong prefix consistency criteria with Consensus is known from [1].

Let us show that we can solve the strong prefix consistency criteria, known to be equivalent to Consensus [1], using $\mathcal{P}_{EP}$. To do so, we consider the following transformation from the protocol $\mathcal{P}_{EP}$. To make an append() operation, processes simply use the append() operation provided by $\mathcal{P}_{EP}$. But, for the the read() operation, processes use the read() operation provided by $\mathcal{P}_{EP}$ to obtain a chain and prune the last dis blocks from it before returning the remaining chain. Note that if there are less than dis blocks, processes then return the genesis block.

Let us show that this modified protocol solves the strong prefix consistency. For this, we need to show that the following properties are satisfied:

- **Chain validity:** The chain validity property is still satisfied by pruning dis blocks.
- **Chain integrity:** The chain integrity property is still satisfied by pruning dis blocks.
- **Strong prefix:** The strong prefix property follows from the known bounded displacement and the removal of the last dis blocks. Indeed, if we remove the last dis blocks, then for any two read() operations, then the first read() returns a prefix of the second read() operation.
- **Ever growing tree:** The ever growing tree property is still satisfied by pruning dis blocks.

For the other direction, we can build a solution to known bounded displacement eventual prefix using a solution for the strong prefix consistency criteria. This trivially solves the eventual known bounded displacement prefix property with $dis = 0$. □

If we consider known bounded displacement eventual prefix then we have strong prefix (we can read a chain and remove the last dis blocks to be sure that what we read is final). Thus, since strong prefix requires Consensus and Consensus in impossible in an asynchronous system (with a single failure) then, known bounded displacement eventual prefix is impossible to solve in an asynchronous system.

3.2 Unknown Bounded Displacement and Eventual Consensus

Eventual Strong prefix and Unknown Bounded Displacement. If we have an unknown bounded displacement, then eventual prefix, combined with such property is stronger than Eventual Consensus. To show that, we first show its equivalence with eventual strong prefix. Later we recall the Eventual Consensus problem (with a small modification of the Validity property, to make it suits to the blockchain context) and finally we show that unknown bounded displacement eventual prefix is stronger than Eventual Consensus.

Theorem 2. *Unknown bounded displacement eventual prefix is equivalent to eventual strong prefix.*

Proof. Let $\mathcal{P}$ be a protocol solving the unknown bounded displacement eventual prefix and let us show that we can solve the eventual strong prefix property. To do so, we consider the following modification to the protocol $\mathcal{P}$. To make an $\mathsf{append}()$ operation, processes simply use the $\mathsf{append}()$ operation provided by $\mathcal{P}$. But, for $\mathsf{read}()$ operation, processes use the $\mathsf{read}()$ operation to obtain a chain and prune the second half of the returned chain before returning the remaining half of the chain.

Let us show that this modified protocol solves the eventual strong prefix property. For this, we need to show that the following properties are satisfied:

- **Chain validity:** The chain validity property is still satisfied by pruning half the chain.
- **Chain integrity:** The chain integrity property is still satisfied by pruning half the chain.
- **Eventual Strong prefix:** The eventual strong prefix property follows from the unknown bounded displacement and the removal of the second half of the chain. Indeed, if we remove the second half of the chain, then eventually for any two $\mathsf{read}()$ operations, then the first $\mathsf{read}()$ returns a prefix of the second $\mathsf{read}()$ operation. Indeed, since we remove a growing number of blocks, eventually we remove more than dis blocks and obtain chains such that one is the prefix of the other.
- **Ever growing tree:** The ever growing tree property is still satisfied by pruning half the chain.

For the other direction, let us consider a protocol $\mathcal{P}$ solving the eventual strong prefix property and let us show that it solves the unknown bounded displacement eventual prefix. The property of eventual strong prefix clearly implies the eventual prefix property. Let $\mathsf{displacement}(b_1, b_2)$ be the function that takes two blockchains b_1 and b_2 and returns the number of blocks needed to prune b_1 and obtain b'_1 such that $b'_1 \sqsubseteq b_2$. Let us show that $\exists dis \in \mathbb{N}, \forall r_{rsp}, r'_{rsp} \in E^2, r \nearrow r', \mathsf{displacement}(r_{rsp}/bc, r'_{rsp}/bc) < dis$. Assume by contradiction that this property is not satisfied, then it implies that for any dis, there exists a couple of reads with a greater displacement than dis. This implies that the eventual strong prefix property is not satisfied which leads to a contradiction, hence eventual strong prefix property implies unknown bounded displacement eventual prefix. □

Eventual Consensus. The eventual consensus [9] (EC) abstraction captures eventual agreement among all participants. It exports, to every process p_i, operations $\mathsf{proposeEC}_1, \mathsf{proposeEC}_2, \ldots$ that take multi-valued arguments (correct processes propose valid values) and return multi-valued responses. Assuming that, for all $j \in \mathbb{N}$, every process invokes $\mathsf{proposeEC}_\mathsf{j}$ as soon as it returns a response to $\mathsf{proposeEC}_{\mathsf{j}-1}$, the abstraction guarantees that, in every admissible run, there exists $k \in \mathbb{N}$ and a predicate $P_{EC}()$, such that the following properties are satisfied:

- **EC-Termination** Every correct process eventually returns a response to $\mathsf{proposeEC}_\mathsf{j}$ for all $j \in \mathbb{N}$.
- **EC-Integrity** No process responds twice to $\mathsf{proposeEC}_\mathsf{j}$ for all $j \in \mathbb{N}$.
- **EC-Validity** Every value returned to $\mathsf{proposeEC}_\mathsf{j}$ is valid with respect to $P_{EC}()$.
- **EC-Agreement** No two correct processes return different values to $\mathsf{proposeEC}_\mathsf{j}$ for all $j \geq k$.

Theorem 3. *Eventual strong prefix is stronger than Eventual Consensus.*

Proof. We show that it exists a protocol $\mathcal{P}_{\mathcal{EC}}$ to solve Eventual Consensus starting from a protocol $\mathcal{P}_{\mathcal{ESP}}$ for eventual strong prefix. We do the transformation as follows. Every correct process p invokes $\mathsf{proposeEC}_\mathsf{j}$ for all $j \in \mathbb{N}$. We impose that the validity predicate $P()$ of the blocktree ADT (see Section 2) be equal to predicate $P_{EC}()$. When a correct process invokes the $\mathsf{proposeEC}_\mathsf{j}(v)$ operation of $\mathcal{P}_{\mathcal{EC}}$, for any $j \in \mathbb{N}$, then it directly invokes the $\mathsf{append}(v)$ operation of $\mathcal{P}_{\mathcal{ESP}}$. After that, p invokes a sequence of $\mathsf{read}()$ operations up to the moment it returns a chain bc such that $bc[j] \neq \bot$. Process p then returns chain bc as decision for $\mathsf{proposeEC}_\mathsf{j}(v)$ and triggers the next operation $\mathsf{proposeEC}_{\mathsf{j}+1}(v')$.

Let us show that protocol $\mathcal{P}_{\mathcal{EC}}$ solves the Eventual Consensus.

- **EC-Termination** This property is guaranteed by the ever growing tree property.
- **EC-Integrity** This property follows directly from the transformation.
- **EC-Validity** This property follows by construction and the chain validity property, since $P() = P_{EC}()$.
- **EC-Agreement** This property follows by the eventual strong prefix property, that guarantees that there exists a $\mathsf{read}()$ operation r such that, all the subsequent ones, return blockchains that are each prefix of the other one. In other words, eventually there is agreement on the value contained in $bc[j]$. This implies that there exists k for which all $\mathsf{proposeEC}_\mathsf{j}$ with $j > k$ returns the same value to all correct processes.

□

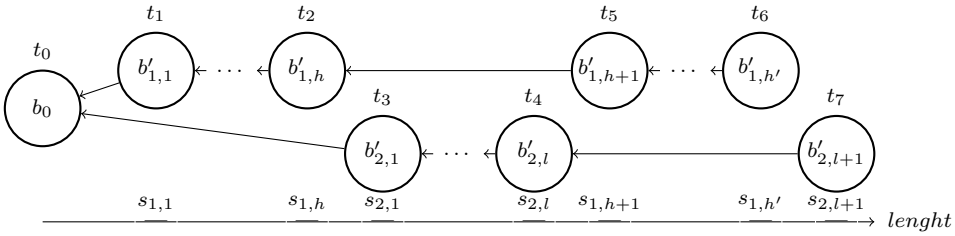

Figure 1: A blocktree generated by two processes. On the x-axis the longest chain value of each chain at different time instants (from the root to the current leaf) and the relationships between those values.

4 Eventual Prefix solutions

In this section we assume that processes are equipped with an Oracle Θ_P or $\Theta_{F,k}$ (as defined in [1]) to access the blockchain, i.e., to read its status (*getValidBlock*() operation) and append a new block to it (*setValidBlock*() operation). Briefly, oracles are abstraction that encapsulate the creation of valid blocks and since only valid blocks can be appended to the blocktree, it follows that oracles grant the access to the blocktree. Oracle $\Theta_{F,k}$ also owns a synchronization power to control the number of forks (k), in terms of branches of the blocktree from a given block. On the other hand Θ_P poses no constraints on the number of blocks that can be appended to the same block, i.e., it allows forks and it poses no bound on the number of blocks in a fork (more details can be found in in [1]).

4.1 Impossibility of Eventual Prefix with longest chain rule

In the following we prove that we cannot provide eventual prefix consistency in the shared-memory model if the append selection function follows the the longest chain rule even when equipped with $\Theta_{F,2}$.

We consider a crash-prone asynchronous system with a finite number of processes and an append selection function f equipped with $\Theta_{F,2}$ that deterministically selects a chain with the longest chain, and in case of a tie selects a chain based on the lexicographical order. We show that it is impossible to implement a blockchain that satisfies eventual prefix consistency for this append selection function.

Note that such a selection function is used by many blockchain systems. In particular in proof-of-work systems such as Bitcoin, chains are selected as the longest chain while in Ethereum chains are selected using the chain with greatest weight, both captured by the selection of chains according to the longest chain.

Theorem 4. *It is impossible to build a blockchain that guarantees eventual prefix consistency when the* append *operation is based on the longest chain rule and is equipped with* $\Theta_{F,2}$.

Proof. Intuitively, the impossibility follows from the fact that with the longest chain selection, races can occur between different branches in the tree. We show that as forks may occur, we can create two infinite branches sharing only the root. Alternatively, one or the other branch constitutes the longest chain and append operations selects chains from each branch alternatively. This is enough to show that the only

common prefix that is returned is the root, hence, violating the eventual prefix consistency.

Let p_1 and p_2 be two processes trying to append infinitely many blocks.

At time t_0, for both p_1 and p_2, the *update_view* of bt equals b_0, thus both call $getValidBlock(b_0, b_{i,1}) = b'_i$, where $i = 1$ for p_1 and $i = 2$ for p_2. At time $t_1 > t_0$, p_1 and p_2 are poised to call $setValidBlock(b_0, b'_{i,1})$. Let p_2 be slower than p_1, thus p_1 cannot distinguish whether p_2 is slow or crashed. Process p_1 gets the set $\{b'_{1,1}\}$ as result of the function and proceeds to append a new block $b_{1,2}$, i.e., it updates bt's view and applies f on it to select the leaf where to append the new block, in this case the only leaf is $b'_{1,1}$. p_1 calls $getValidBlock(b'_{1,1}, b_{1,2})$ which returns $\{b'_{1,2}\}$ and hereafter at time $t_2 > t_1$ it is poised to call $setValidBlock(b'_{1,1}, b'_{1,2})$.

We can continue in the same way up to time $t_3 \geq t_2$, the moment in which the function $setValidBlock(b_0, b'_{2,1})$ returns $\{b'_{1,1}, b'_{2,1}\}$ to p_2. We assume that at time t_3, p_1 built a chain from the root b_0 to $b'_{1,h}$ and is poised to append $b'_{1,h+1}$. The chain from b_0 to $b'_{1,h}$ has a length $s_{1,h}$ and the chain b_0 to $b'_{2,1}$ has a length $s_{2,1}$. W.l.o.g., we assume that $s_{1,h} < s_{2,1}$ and $s_{1,h+1} > s_{2,1}$. At time $t_4 > t_3$, p_2 updates its view of bt and f applied to it selects the chain from b_0 to $b'_{2,1}$, as $s_{1,h} < s_{2,1}$, to append the new block. We can make it append blocks until it is poised to call a $setValidBlock(b'_{2,l}, b'_{2,l+1})$ such that $s_{2,l} < s_{1,h+1}$ and $s_{2,l+1} > s_{2,h+1}$ (cf. Figure 1). We can continue to create two infinite branches sharing only the root. A read operation can alternatively return a chain from each branch, depending on the time (cf. Figure 1), e.g., from b_0 to $b'_{1,h}$ at time t_2 and from b_0 to $b'_{2,l}$ at time t_4. Thus, the common prefix never increases, and so, eventual prefix consistency is not satisfied. □

4.2 Asynchronous solution to eventual prefix with an unbounded number of Byzantine processes

We consider an asynchronous system with a possibly infinite set of processes which can append infinitely many blocks, and processes can be affected by Byzantine failures. Each process has a unique identifier that can be verified using signatures. We show that in that setting it is possible to build a blockchain that satisfies eventual prefix consistency. This is done by using the oracle Θ_P which emulates a shared-memory model.

Algorithm 1: Append procedure

```
   Input: Blockchain bt
1  bt.update_view()
2  if Θ_P.getValidBlock(f_a(bt)) then
3  |  Θ_P.setValidBlock(f_a(bt))
4  |  return ⊤
5  end if
6  else
7  |  return ⊥
8  end if
```

Algorithm 2: Read procedure

Input: Blockchain bt

1 $bt.update\_view()$

2 **return** $f_r(bt)$

Algorithm The main idea of the algorithm consists on using a local selection function for append operations.

The append select function f_r selects a chain by going from the root to a leaf, choosing at each fork the edge to the child with the lowest identifier, given some arbitrary total order on blocks. The read selection function f_r is the same as f_a.

To perform an append() operation, processes update the current view of the bt's state and apply the *getValidBlock* operation of Θ_P for the last block in $f_a(bt)$. Finally, they apply the *setValidBlock* to append it to bt. For read() operations, processes return the chain selected by f_r on the bt returned by *update_view*.

Proof sketch Chain validity and chain integrity properties are satisfied by Oracle Θ_P. The ever-growing tree property relies on the fact that each fork has a finite number of blocks since there are finitely many processes and each (Byzantine or correct) process can contribute with at most one block per parent as multiple children created by the same process are ignored by f_a. Thus, eventually, new blocks contribute to the growth of the tree. The eventual prefix property follows from the definition of f_a. For any b in bt, let t_b be the time after which no process contributes to the same block b. All correct processes trigger the append operation on chains containing the non-ignored child with the longest chain. By induction, it creates an ever-growing prefix selected by f_a. As read chains are selected according to the append selection function, it provides the eventual prefix property.

4.3 Eventual synchronous solution for Unknown Bounded Displacement Eventual Prefix

In this Section we discuss the impossibilities and possibilities of unknown bounded displacement eventual prefix consistency. In particular, we show that it is unsolvable in an asynchronous shared-memory or message-passing system if at least one process fails with Byzantine failures. Interestingly, if we consider the previous result with known bounded displacement, the impossibility holds even in an eventually synchronous system model with more than one third Byzantine processes (with not surprise, given the equivalence with the Consensus problem, see Theorem 1). Finally we prove that unknown bounded displacement eventual prefix consistency is solvable in an eventual synchronous message-passing system with a majority of correct processes.

Theorem 5. *There does not exist any solution that solves unknown bounded displacement eventual prefix in an asynchronous system with at least one Byzantine process.*

Proof. The proof follows from the relationship between the unknown bounded displacement eventual prefix which is stronger than the Eventual Consensus problem

(cf. Theorem 3), which is equivalent to the Leader Election problem [9] which cannot be solved in an asynchronous system with at least one Byzantine processes [18]. □

Theorem 6. *There does not exist any solution that solves known bounded displacement eventual prefix in an eventual synchronous system with more than one third of Byzantine faulty processes.*

Proof. The proof follows from the equivalence between known bounded displacement eventual prefix and Consensus (cf. Theorem 1), which is unsolvable in a synchronous (and thus also in an eventually synchronous) system with more than one third of Byzantine faulty process [13]. □

Theorem 7. *There exists a solution that solves unknown bounded displacement eventual prefix in an eventual synchronous system with a majority of correct processes.*

In order to prove the existence of a solution, we start from an existing solution, Streamlet [6], that guarantees strong prefix consistency under the assumption of less than a third of byzantine processes and eventual synchrony with a known message delay Δ. We weaken both of these assumptions to provide a solution to the eventual strong prefix consistency criteria. In particular, we assume only a majority of correct processes, we do not explicitly use Δ and consider a slightly modified version of the protocol. In the following we first describe Streamlet and the discuss the modifications before providing the proof.

Streamlet protocol. The Streamlet protocol works in a partially synchronous system with a known message delay Δ and a finite set of n processes. In particular, before the Global Stabilisation Time (GST), message delays can be arbitrary; however, after GST, messages sent by correct processes are guaranteed to be received by correct processes within Δ time. The following assumption holds: Δ-bounded assumption during periods of synchrony, when an honest node sends a message at time t, an honest recipient is guaranteed to receive it by time $max(GST, t + \Delta)$. [1]

In Streamlet [6], each epoch, composed of 2Δ, has a designated leader chosen at random by a publicly known hash function. The protocol works as follows:

- **Propose-Vote.** In every epoch:
  - The epoch's designated leader proposes a new block extending from the longest notarized chain it has seen (if there are multiple, break ties arbitrarily). The notion "notarized" is defined below.
  - Every process votes for the first proposal they see from the epoch's leader, as long as the proposed block extends from (one of) the longest notarized chain(s) that the voter has seen. A vote is a signature on the proposed block.
  - When a block gains votes from at least 2n/3 distinct processes, it becomes notarized. A chain is notarized if its constituent blocks are all notarized.

[1] Notice that, in Streamlet [6] there is not the notion of time but of round, which denotes a basic unit of time.

- **Finalize.** Notarized does not mean final. If in any notarized chain, there are three adjacent blocks with consecutive epoch numbers, the prefix of the chain up to the second of the three blocks is considered final. When a block becomes final, all of its prefix must be final too.

We propose protocol $\mathcal{S}$ with the following modifications to Streamlet. First, we only require that a block gains votes from a majority of distinct processes to become notarized, which means that forks can occur. The second modification goes deeper: if a fork occurs, then it is possible to detect Byzantine processes and to exclude them from the voters. This is done as follows. When, two conflicting chains are finalized, that is two finalized chains that are not the prefix of one another, then processes look for inconsistent blocks. That is, two notarized blocks b, b' are inconsistent with one another if one of the following two conditions hold:

- Cond. 1 b and b' share the same epoch.
- Cond. 2 b (resp. b') that corresponds to a higher epoch, $b.epoch < b'.epoch$ (resp. $b'.epoch < b.epoch$), with a strictly smaller height, $b'.height < b.height$ (resp. $b.height < b'.height$), than the other block.

If a process votes for blocks inconsistent with one another then it is detected as Byzantine.

Proof. Let us show that $\mathcal{S}$ is a solution for eventual strong prefix. Let us first show that when a fork occurs, then we detect at least a Byzantine process. For this let us show that it implies the existence of two inconsistent blocks and that the intersection of voters of the two blocks is not empty as two majority intersect. In the following we show that voting for two inconsistent blocks is a Byzantine failure.

Let us first show that voting for two inconsistent blocks b and b' is a Byzantine failure. If the two blocks are inconsistent for Cond. 1, then the intersecting voters are Byzantine as correct processes vote only once per epoch. It follows that if a process q votes for b and b' then q is Byzantine. If the two blocks are inconsistent for Cond. 2, then the intersecting voters are Byzantine as correct processes vote only for blocks extending one of the longest notarized chains. That is, if a correct process p votes for b it means that b is extending a notarized block b_{pred} that is of height $b.height - 1$, therefore p cannot vote for a block b' later on with a height strictly smaller than $b.height$ because it needs to extend one of the longest notarized chain. It follows that if a process q votes for b and b' then q is Byzantine.

Let us now show that when a fork occurs we must have two inconsistent blocks. Indeed, if there is a fork then we have two sequences of three adjacent blocks with consecutive epochs, b_1, b_2, b_3 and b'_1, b'_2, b'_3 (by construction, given the finalization rule). If no blocks share the same epoch number then we can assume w.l.o.g. that $b_3.epoch < b'_1.epoch$. Let block b' be the block with the smallest height that is in the prefix of b'_3 such that it has a greater epoch than b_1 (such block always exists as b'_1 satisfies those conditions). Either $b'.height < b_3.height$ meaning that b' is inconsistent with b_3 or else, $b'.height \geq b_3.height$ meaning that the predecessor of b' is inconsistent with b_1. Indeed, the predecessor of b' has a strictly smaller height than b_1 and by assumption has a smaller epoch number than b_1. Hence there is always a couple of inconsistent blocks in a fork.

Let us now conclude our proof that we solve the eventual strong prefix property. If a fork occurs, then each correct process eventually detects at least one Byzantine process and ignore its votes, hence, we have a finite number of forks as we have a finite number of Byzantine processes, hence eventually there is always a single chain that is finalized. As there is a majority of correct processes, we remain live as in the original Streamlet Protocol. We also inherit its properties for finalizing blocks eventually when synchrony is reached. □

5 Conclusion

In this work we closed the gap between eventual prefix and strong prefix consistency. We firstly addressed the question: "can we design an asynchronous deterministic blockchain solution to solve bounded displacement eventual prefix ?". We formally showed that the answer is "no". On the positive side, we provided for the first time (i) a solution to eventual strong prefix with a majority of correct processes and (ii) an asynchronous solution to eventual prefix with an unlimited number of Byzantine processes.

References

[1] E. Anceaume, A. D. Pozzo, R. Ludinard, M. Potop-Butucaru, and S. Tucci Piergiovanni. Blockchain abstract data type. *CoRR*, 2018.

[2] E. Anceaume, A. D. Pozzo, R. Ludinard, M. Potop-Butucaru, and S. Tucci Piergiovanni. Blockchain abstract data type. In *SPAA*, pages 349–358, 2019.

[3] E. Androulaki and et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proc. EuroSys*, 2018.

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

[5] C. Cachin. Blockchain - From the Anarchy of Cryptocurrencies to the Enterprise (Keynote Abstract). In *Proc. of the OPODIS International Conference*, 2016.

[6] B. Y. Chan and E. Shi. Streamlet: Textbook streamlined blockchains, 2020.

[7] J. Chen and S. Micali. Algorand: A secure and efficient distributed ledger. *Theor. Comput. Sci.*, 2019.

[8] T. Crain, V. Gramoli, M. Larrea, and M. Raynal. (leader/randomization/signature)-free byzantine consensus for consortium blockchains. *CoRR*, abs/1702.03068, 2017.

[9] S. Dubois, R. Guerraoui, P. Kuznetsov, F. Petit, and P. Sens. The weakest failure detector for eventual consistency. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 375–384, 2015.

[10] J. A. Garay, A. Kiayias, and N. Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Proc. EUROCRYPT International Conference*, 2015.

[11] R. Guerraoui, P. Kuznetsov, M. Monti, M. Pavlovič, and D.-A. Seredinschi. The consensus number of a cryptocurrency. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, page 307–316, New York, NY, USA, 2019. Association for Computing Machinery.

[12] M. Herlihy. Blockchains and the future of distributed computing. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC '17, pages 155–155, New York, NY, USA, 2017. ACM.

[13] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 1982.

[14] B. Liskov and S. Zilles. Programming with abstract data types. *ACM SIGLAN Notices*, 9(4), 1974.

[15] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *www.bitcoin.org*, 2008.

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

[17] R. P. R., L. Seeman, and A. Shelat. Analysis of the blockchain protocol in asynchronous networks. In *Proc. of the EUROCRYPT International Conference*, 2017.

[18] M. Raynal. Eventual leader service in unreliable asynchronous systems: Why? how? In *Sixth IEEE International Symposium on Network Computing and Applications (NCA 2007)*, pages 11–24. IEEE, 2007.

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