On finality in blockchains

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

On Finality in Blockchains

Emmanuelle Anceaume
CNRS, Univ Rennes, Inria, IRISA, France

Antonella Del Pozzo
Paris-Saclay, CEA, List, France

Thibault Rieutord
Paris-Saclay, CEA, List, France

Sara Tucci-Piergiovanni
Paris-Saclay, CEA, List, France

Abstract

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. 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.

The quest for "good" deterministic finality properties for blockchains is still in its infancy, though. Our motivation is to provide a thorough study of several possible deterministic finality properties and explore their solvability. This is achieved by introducing the notion of bounded revocation, which informally says that the number of blocks that can be revoked from the current blockchain is bounded. Based on the requirements we impose on this revocation number, we provide reductions between different forms of eventual finality, Consensus and Eventual Consensus. From these reductions, we show some related impossibility results in presence of Byzantine processes, and provide non-trivial results. In particular, we provide an algorithm that solves a weak form of eventual finality in an asynchronous system in presence of an unbounded number of Byzantine processes. We also provide an algorithm that solves eventual finality with a bounded revocation number in an eventually synchronous environment in presence of less than half of Byzantine processes. The simplicity of the arguments should better guide blockchain designs and link them to clear formal properties of finality.

2012 ACM Subject Classification

Keywords and phrases Blockchain specification, Finality

Digital Object Identifier 10.4230/LIPIcs...

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. 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. Assuming partially synchronous environments, most of the permissioned blockchains satisfy the deterministic form of immediate consistency, as for example Red Belly blockchain [9] and Hyperledger Fabric blockchain [2]. The probabilistic form of immediate consistency is typically achieved by permissionless pure proof-of-stake blockchains such as Algorand [8].

Unlike immediate finality, eventual finality only 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 [21] and Ethereum [27] 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. More recently, a huge effort has been devoted to propose alternatives to the energy-wasting proof-of-work method of Bitcoin and Ethereum. These proof-of-stake blockchains (e.g. [17, 22, 13, 16]) offer as well a form of eventual finality. More broadly, all these solutions favor availability (or progress) under adversarial conditions, therefore they do not rely on Byzantine Consensus. This implies that it is admitted that a blockchain may lose consistency by incurring a fork, which is the presence of multiple chains at different processes. The heart of these solutions is then a reconciliation mechanism, which is always available to recover from a fork. Reconciliation typically consists in a local deterministic rule selecting a chain among the different alternatives available. In Bitcoin for instance any participant is able to reconcile the state following the "longest" chain rule. Once a winner chain is chosen, the other alternatives are revoked, as such all the blocks belonging to them. It is important to stress that the design of these blockchains aims at being resistant to adversarial participants creating alternative chains on purpose in synchronous environments. This means that during reconciliation all the candidate chains available at one honest participant are also available at all other honest participants. This allows us to compute finalisation probabilistic guarantees, such as the one in Bitcoin that says that it is computationally hard to revoke a block followed by six other blocks in presence of no more than 10% of Byzantine participants (selfish attack). Real large-scale distributed systems, however, can hardly be synchronous. Network effects make the moment at which all honest processes observe the same set of candidate chains unknown. The asynchrony effect might render the reconciliation rule and finalisation guarantees unsure, or simply extremely inefficient, for example by considering a block as finalised after one or more days. To solve this problem a number of permissionless blockchain projects are investigating how to add "finality gadgets" (e.g., [5, 26]) to proof-of-work or proof-of-stake blockchains, which means seeking additional mechanisms or protocols to reach "better" finality properties in network adversarial settings. The hope is to find ways to get deterministic finality by periodically running finality gadgets. For the time being, the only way that has been concretely pursued is to add Byzantine Consensus – e.g. Tenderbake [4] adds Byzantine Consensus to the existing proof-of-stake method assuring deterministic finality to each block followed by other two blocks. How to add mechanisms that do not resort to Consensus, however, is an intriguing and open question, related to the finality properties one would like to guarantee.

The quest for "good" deterministic finality properties for blockchains is still in its infancy, though. Our motivation is to provide a thorough study of several possible finality properties and explore their solvability. In doing so, we are particularly intrigued by answering the question, what lies between eventual finality and immediate finality?

To this aim we introduce the notion of bounded revocation, which informally says that the number of blocks that can be revoked from the current blockchain is bounded. Providing solutions that guarantee deterministic bounded revocation reveals to be an important crux in the construction of blockchains. We thus provide rigorous definitions for the weakest form of eventual finality $\mathcal{EF}^{\star}$, which does not guarantee any bound on the number of blocks that can be revoked from the blockchain at any given fork, and two stronger forms: $\mathcal{EF}^{\Diamond c}$, in which revocation is bounded but unknown, and $\mathcal{EF}^{c}$ in which revocation is bounded and known. Intuitively, if $\mathcal{EF}^{c}$ holds, processes revoke at most a constant number c of blocks from the current blockchain at each reconciliation, while if $\mathcal{EF}^{\Diamond c}$ holds, processes revoke at most a constant number of c blocks from the current chain only eventually, i.e., after a finite but unknown number of reconciliations.

The rigorous formalisation of these properties enable us to easily show that solutions that guarantee $\mathcal{EF}^c$ are equivalent to Consensus, while solutions that guarantee $\mathcal{EF}^{\Diamond c}$ are not weaker than Eventual Consensus, an abstraction that captures eventual agreement among all participants. From these reductions, we show some related impossibility results in presence of Byzantine processes. Beside reductions and related impossibilities, we propose the following non-trivial results:

- $\mathcal{EF}^\star$ cannot be achieved in an asynchronous system if the reconciliation rule follows the "longest" chain rule (Theorem 13). This implies that the reconciliation rule, used in current blockchains, to provide probabilistic finality in synchronous settings cannot guarantee that participants will eventually converge to a stable prefix of the chain in asynchronous settings.
- A solution that guarantees $\mathcal{EF}^\star$ in an asynchronous system with a possibly infinite set of processes which can append infinitely many blocks. This novel solution is strikingly simple and tolerant to an unbounded number of Byzantine processes (Theorem 14).
- A solution that solves $\mathcal{EF}^{\Diamond c}$ in an eventually synchronous environment in presence of less than half of Byzantine processes (Theorem 15). The central point of our solution is to let correct processes blame each fork on a particular Byzantine process, which can then be excluded from the computation. Weakening the classic requirement of $< 1/3$ to $< 1/2$ Byzantine processes makes such a solution well adapted to large scale adversarial systems. As for the previous one, we are not aware of any such solution in the literature.

We hope that these results will better guide blockchain designs and link them to clear formal properties of finality.

1.0.0.1 Related Work

Formalization of blockchains in the lens of distributed computing has been recognized as an extremely important topic [15]. Garay et al. [11] have been the first to analyze 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 have analyzed the protocol in a synchronous system, while Pass et al. [23] have extended this line of work considering a more adversarial network. Anta et al. [3] have 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 [14] have formalized the notion of cryptocurrency showing that Consensus is not needed.

While probabilistic eventual finality has been widely studied in the context of Bitcoin [11, 6, 23], only a few studies have started to lay the foundations of the computation power of blockchains with deterministic eventual finality consistency. Anceaume et al. [1] have been the first to capture 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 multiple chains can co-exist but the common prefix eventually converges. Interestingly, the authors of [1] show that to solve strong prefix the Consensus abstraction is needed, however they do not address solvability of eventual prefix, which is the focus of this paper.

The paper is organised as follows: Section 2 formally presents the sequential specification of a blockchain and the formalisation of the different finality properties we may expect from a blockchain when concurrently accessed. Section 3 presents reductions between different forms of finality, Consensus and Eventual Consensus. Section 4 first shows why $\mathcal{EF}^\star$ is not solvable in an asynchronous environment when the "longest" chain rule is used, and then presents the algorithms to solve $\mathcal{EF}^\star$ and $\mathcal{EF}^{\Diamond c}$. These algorithms are particularly simple. Finally, Section 5 concludes.

2 Definitions

2.1 Preliminary Definitions

We describe a blockchain object as an abstract data type which allows us to completely characterize a blockchain by the operations it exports [19]. 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 [24]. Prior to presenting the blockchain abstract data type we first recall the formalization used to describe an abstract data type (ADT).

2.1.0.1 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 $\tau : Z \times A \rightarrow Z$ is the *transition function*, specifying the effect of an input on the object state and the map $\delta : Z \times A \rightarrow B$ is the *output function*, specifying the output returned for a given input and an object local state. An 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 depend on the state in which it is called and the output function δ.

2.1.0.2 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;*
- $\Lambda : E \rightarrow \Sigma$ *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 blocktree ADT

We represent a blockchain as a tree of blocks. Indeed, while consensus-based blockchains prevent forks or branching in the tree of blocks, 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).

2.2.1 Sequential Specification of the Blocktree ADT (BT-ADT)

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 $\mathcal{BT}$ be the set of blocktrees, $\mathcal{B}$ 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 (note that in [1], the read() and append() operations are defined with a unique selection function). The read selection $f_r()$ takes as argument the blocktree and returns a chain of blocks, that is a sequence of blocks starting from the genesis block to a leaf block of the blocktree. The chain b_c returned by a read() operation r is called the blockchain, and is denoted by r/bc. 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. Only blocks satisfying some validity predicate P can be appended to the tree. 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 $f_a()$ (denoted by $f_a(bt)^\frown b$). In Bitcoin for instance this predicate embeds the logic to verify that the obtained chain does not contain double spending or overspending transactions. Formally,

▶ **Definition 2.** *(Sequential specification of the Blocktree ADT) The Blocktree Abstract Data Type is the* 6*-tuple* $\mathrm{BT-ADT}=\{A = \{append(b), read()/bc \in \mathcal{BC}\}, B = \mathcal{BC} \cup \{\top, \bot\}, Z = \mathcal{BT}, \xi_0 = b_0, \tau, \delta\}$*, 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*

$$\delta(bt, read()) = f_r(bt)$$

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

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

2.2.2 Concurrent Specification and Consistency Criteria of the BlockTree ADT

The concurrent specification of the 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 the blocktree abstract data type.

We define three consistency criteria for the blocktree, i.e., the *BT eventual finality (EF)*, the *BT immediate finality (IF)* and *BT eventual immediate finality (EIF)*, and the notion of block revocation. This family of consistency criteria combined with the revocation notion provide a comprehensive characterization of what we may expect from blockchains.

▶ **Notation 1.**

- $E(a^*, r^*)$ *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;*
- $\mathsf{length} : \mathcal{BC} \rightarrow \mathbb{N}$ *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* $\mathsf{length}(bc^\frown\{b\}) > \mathsf{length}(bc)$*;*
- $bc \sqsubseteq bc'$ *iff bc prefixes bc'.*
- $bc[i]$ *refers to the i-th block of blockchain bc.*

▶ **Definition 3** (BT Eventual Finality Consistency criterion (EF))**.** *A concurrent history* $H = \langle \Sigma, E, \Lambda, \mapsto, \prec, \nearrow \rangle$ *of a system that uses a BT-ADT verifies the BT eventual finality consistency criterion if the following four 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:**
  $\forall E \in E(a^*, r^*), \forall k \in \mathbb{N}, \exists r \in E : \mathsf{length}(r_{rsp}/bc) > k$.
  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.*

▶ **Definition 4** (BT Immediate Finality Consistency criterion (IF)). *A concurrent history $H = \langle \Sigma, E, \Lambda, \mapsto, \prec, \nearrow \rangle$ of the system that uses a BT-ADT verifies the BT immediate finality consistency criterion if chain validity, chain integrity, ever growing tree (as defined for EF) 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.

▶ **Definition 5** (BT Eventual Immediate Finality Consistency criterion (EIF)). *A concurrent history $H = \langle \Sigma, E, \Lambda, \mapsto, \prec, \nearrow \rangle$ of the system that uses a BT-ADT verifies the BT eventual immediate finality consistency criterion if chain validity, chain integrity, ever growing tree (as defined for EF) 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 revocation

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

▶ **Definition 6.** *c-Bounded revocation*

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

▶ **Notation 2.** For readability reasons, in the following we will simply say *finality* instead of *finality consistency criterion*, i.e., eventual finality consistency criterion will be replaced by eventual finality, and (eventual) immediate finality consistency criterion will be replaced by (eventual) immediate finality.

We can now define the $c-$Bounded Eventual Finality criteria by augmenting the previous consistency criteria with the Bounded revocation property:

▶ **Definition 7.** *$c-$Bounded Eventual Finality criteria*

- $\mathcal{EF}^\star = EF$*, in this case the revocation is unbounded.*
- $\mathcal{EF}^c = EIF$ *combined with c−Bounded revocation, such that c is known a priori.*
- $\mathcal{EF}^{\Diamond c} = EIF$ *combined with c−Bounded revocation where c is unknown but bounded.*

We will show in the following that satisfying $\mathcal{EF}^c$ is equivalent to immediate finality (IF). This is because from any algorithm $\mathcal{P}$ implementing $\mathcal{EF}^c$, if we take the blockchain that is returned by a read provided by $\mathcal{P}$ except for the last c blocks, this guarantees the strong prefix property of IF. Furthermore, $\mathcal{EF}^{\Diamond c}$ boils down to eventual immediate finality (EIF). Indeed as shown later, if we take half of the blockchain returned by a read provided by an algorithm $\mathcal{P}$ implementing $\mathcal{EF}^{\Diamond c}$, this guarantees eventual immediate finality since chains are always growing, and thus the number of removed blocks increases up to reaching c.

In the following section we prove the above-mentioned equivalences more formally and study relationships to known problems such as consensus and eventual consensus.

3 (Eventual) Consensus Reductions

In this section we investigate the impact of the bounded revocation property on the construction of a blocktree satisfying eventual finality. In particular, we show that when the bound c is known, this problem is equivalent to the consensus abstraction, while when unknown, this problem is not weaker than the eventual consensus abstraction [10].

3.1 Known Bounded Revocation and Consensus

▶ **Theorem 8.** *$\mathcal{EF}^c$ is equivalent to Consensus.*

Proof. We first show how to solve immediate finality (IF) given a solution $\mathcal{P}$ for $\mathcal{EF}^c$ and then the reciprocal direction. Indeed, the equivalence between immediate finality and consensus is known from [1]. So let us show that we can solve immediate finality using $\mathcal{P}$. To do so, we consider the following transformation from the protocol $\mathcal{P}$. To make an append() operation, processes simply use the append() operation provided by $\mathcal{P}$. But, for the the read() operation, processes use the read() operation provided by $\mathcal{P}$ to obtain a chain and prune the last c blocks from it before returning the remaining chain. Note that if there are less than c blocks, processes then return the genesis block.

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

- **Chain validity:** The chain validity property is still satisfied by pruning the last c blocks.
- **Chain integrity:** The chain integrity property is still satisfied by pruning the last c blocks.
- **Strong prefix:** The strong prefix property follows from the known bounded revocation property and the removal of the last c blocks. Indeed, if we remove the last c 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 the last c blocks.

For the other direction, we can build a solution to $\mathcal{EF}^c$ using a solution for immediate finality (IF). This trivially solves $\mathcal{EF}^c$ with $c = 0$. ◀

From Theorem 8 immediately follows the following impossibility result:

▶ **Theorem 9.** *There does not exist any solution that solves $\mathcal{EF}^c$ in an eventual synchronous system with more than $n/3$ Byzantine processes, where n is the number of processes participating to the algorithm.*

Proof. The proof follows from the equivalence between $\mathcal{EF}^c$ and Consensus (cf. Theorem 8), which is unsolvable in a synchronous (and thus also in an eventually synchronous) system with more than one third of Byzantine processes [18]. ◀

3.2 Unknown Bounded Revocation and Eventual Consensus

In this section we show that $\mathcal{EF}^{\diamond c}$ is not weaker than eventual consensus. We first show its equivalence with eventual immediate finality (EIF). Later we recall the eventual consensus problem with a small modification of the validity property to make it suitable to the blockchain context and show that eventual immediate finality is not weaker than eventual consensus.

▶ **Theorem 10.** *$\mathcal{EF}^{\diamond c}$ is equivalent to eventual immediate finality.*

Proof. Let $\mathcal{P}_1$ be a protocol solving $\mathcal{EF}^{\Diamond c}$ and let us show that we can solve eventual immediate finality. To do so, we consider the following modification to the protocol $\mathcal{P}_1$. To make an $\mathsf{append}()$ operation, processes simply use the $\mathsf{append}()$ operation provided by $\mathcal{P}_1$. But, for a $\mathsf{read}()$ operation, processes use the $\mathsf{read}()$ operation provided by $\mathcal{P}_1$ 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 eventual immediate finality. For this, we need to show that the following properties are satisfied:

- **Chain validity:** The chain validity property is still satisfied by pruning half of the chain.
- **Chain integrity:** The chain integrity property is still satisfied by pruning half of the chain.
- **Eventual strong prefix:** The eventual strong prefix property follows from the unknown bounded revocation property 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 at least c 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 of the chain.

For the other direction, let us consider a protocol $\mathcal{P}_2$ solving the eventual immediate finality and let us show that it solves $\mathcal{EF}^{\Diamond c}$. The property of eventual strong prefix property clearly implies the eventual prefix property. Let $\mathsf{revocation}(b_1, b_2)$ be the function that takes two blockchains b_1 and b_2 and returns the number of blocks needed to prune b_1 to obtain a chain b_1' such that $b_1' \sqsubseteq b_2$. Let us show that $\exists c \in \mathbb{N}, \forall r_{rsp}, r'_{rsp} \in E^2, r \nearrow r', \mathsf{revocation}(r_{rsp}/bc, r'_{rsp}/bc) < c$. Assume by contradiction that this inequality is not satisfied, then it implies that for any c, there exists a couple of reads with a greater revocation than c. This implies that the eventual strong prefix property is not satisfied, which leads to a contradiction. Hence eventual immediate finality implies $\mathcal{EF}^{\Diamond c}$. Putting all together, we have shown that eventual immediate finality is equivalent to $\mathcal{EF}^{\Diamond c}$. ◀

The eventual consensus (EC) abstraction [10] 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 predicate P_{EC}.
- **EC-Agreement.** No two correct processes return different values to $\mathsf{proposeEC}_\mathsf{j}$ for all $j \geq k$.

▶ **Theorem 11.** *Eventual immediate finality is not weaker than eventual consensus.*

Proof. We show that there exists a protocol $\mathcal{P}_1$ to solve eventual consensus starting from a protocol $\mathcal{P}_2$ that solves eventual immediate finality. 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_1. When a correct process p invokes the $\mathsf{proposeEC_j}(v)$ operation of $\mathcal{P}_1$, for any $j \in \mathbb{N}$, then p executes the following sequence of three steps: *(i)* it invokes the $\mathsf{append}(v)$ operation of $\mathcal{P}_2$, then *(ii)* it invokes a sequence of $\mathsf{read}()$ operations up to the moment the $\mathsf{read}()$ returns a chain bc such that $bc[j] \neq \perp$, and finally *(iii)* p returns chain bc as decision for $\mathsf{proposeEC_j}(v)$ and triggers the next operation $\mathsf{proposeEC_{j+1}}(v')$.

Let us show that protocol $\mathcal{P}_1$ solves 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 predicate P equals to predicate P_1.
- **EC-Agreement** This property follows by the eventual strong prefix property, which guarantees that there exists a $\mathsf{read}()$ operation r such that, all the subsequent ones return blockchains that are each prefix of the following one. In other words, eventually there is agreement on the value contained in $bc[j]$. This implies that there exists k for which all $\mathsf{proposeEC_j}$ with $j \geq k$ return the same value to all correct processes.

◀

▶ **Theorem 12.** *There does not exist any solution that solves $\mathcal{EF}^{\Diamond c}$ in an asynchronous system with at least one Byzantine process.*

Proof. The proof follows from the relationship between the $\mathcal{EF}^{\Diamond c}$ and eventual immediate finality (EIF). EIF is not weaker than the eventual consensus problem (cf. Theorem 11), which is equivalent to the leader election problem [10] which cannot be solved in an asynchronous system with at least one Byzantine process [25]. ◀

4 Eventual Finality Solutions

In this section we first show the impossibility of solving $\mathcal{EF}^\star$ when the append operation, in case of forks, selects the "longest" chain. We then provide the first solution to $\mathcal{EF}^\star$ with an unbounded number of Byzantine processes using an alternative selection rule.

4.1 Impossibility of Eventual Finality with the Longest Chain Rule

In the following we prove that we cannot provide $\mathcal{EF}^\star$ if, in case of forks, the append selection function $f_a()$ follows the longest chain rule, i.e., returns the longest chain of the blockchain tree. To show this impossibility, we consider a scenario in which the occurrence of any fork produces at most two alternative chains (this is often referred to as a branching factor of 2). We consider a finite number of processes and an append selection function f_a that in case of forks deterministically selects the longest chain, i.e., the chain with the largest number of blocks (the length is thus a monotonically increasing function on prefixes), and in case of a tie selects the chain whose last block is the smallest (in the lexicographical order). We show that it is impossible to guarantee $\mathcal{EF}^\star$ for such append selection function f_a.

Note that such a selection function is used by many blockchain systems. In proof-of-work systems such as Bitcoin, chains are selected as the chain with the greater number of blocks (actually this corresponds to the heaviest one by considering the difficulty) while in Ethereum chains are selected using the chain with greatest weight, both captured by the selection of chains according to the longest chain. In proof-of-stake systems like EOS [13] or Tezos [12] the same rule is also applied.

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

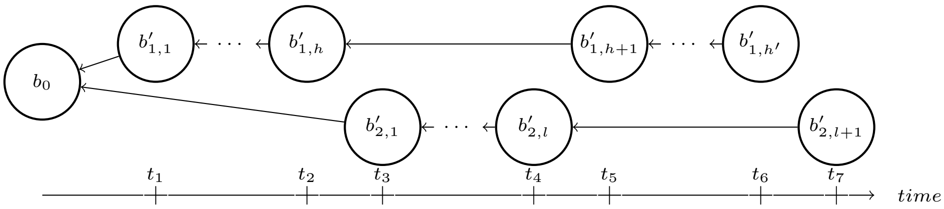

Figure 1 A blocktree generated by two processes. On the x-axis the longest chain value of each chain at different time instants (from the root to the current leaf) and the relationships between those values.

can create two infinite branches sharing only the root. One or the other branch constitutes alternatively the longest chain and append operations select chains from each branch alternatively. This is enough to show that the only common prefix that is returned is the root hence, violating eventual finality.

Obviously, this impossibility result holds only when blocks are not created by running a consensus algorithm. When consensus is employed, immediate finality can be assured, and no fork will ever occur. In this case the append operation will return the longest chain by default.

To capture the synchronisation power of the system, we introduce an oracle that regulates the number of appended children from a same parent. The same approach has been proposed in [1]. The branching factor of an oracle is the maximal number of children that can be appended to a block. The oracle is the only generator of valid blocks. It owns a synchronization power equal to Consensus if its branching factor is equal to 1.

The oracle grants access to the blocktree as a shared object, through the following three operations: update_view() returns the current state of the blocktree; getValidBlock(b_i, b_j) returns a valid block b'_j, constructed from b_j, that can be appended to block b_i, where b_i is already included in the blocktree; and setValidBlock(b_i, b'_j) appends the valid block b'_j to b_i, and returns $\top$ when the block is successfully appended and $\bot$ otherwise.

▶ **Theorem 13.** *It is impossible to guarantee $\mathcal{EF}^\star$ if the* append *operation is based on the longest chain rule in an asynchronous environment.*

Proof. In the proof we consider the stronger oracle allowing the occurrence of one fork, i.e., an oracle with branching factor equal to 2. That is, this oracle allows for two valid blocks to be appended to the same parent, afterwards, it shall return $\bot$ to all requests.

Let p_1 and p_2 be two processes trying to append infinitely many blocks. W.l.o.g., we carry out this proof with a length function equal to the number of blocks.

At time t_0, for both p_1 and p_2, the update_view() of bt equals b_0, thus when both apply the append selection function f_a on it to select the leaf where to append the new block, they both get b_0. Then they 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($b_0, b'_{i,1}$). We then let p_1 call setValidBlock($b_0, b'_{1,1}$), which must return $\top$ and hence $b'_{1,1}$ is appended to b_0. Process p_1 then proceeds to append a new block $b_{1,2}$, i.e., after having updated its bt's view, through the update_view() function, p_1 applies the append selection function f_a on it to select the leaf where to append its new block, in this case the only leaf is $b'_{1,1}$. It calls getValidBlock($b'_{1,1}, b_{1,2}$) function which returns $\{b'_{1,2}\}$ and it is poised to call setValidBlock($b'_{1,1}, b'_{1,2}$).

We let p_1 continue to append new blocks until some time t_2 at which it is poised to call setValidBlock($b'_{1,h}, b'_{1,h+1}$), with $h = 1$, such that the length of the chain $b_0, \ldots, b'_{1,h+1}$

would be greater than or would have the same length but a smaller lexicographical order than the chain $b_0, b'_{2,1}$ if $b'_{2,1}$ were already appended to block b_0. Afterwards, at time $t_3 \geq t_2$, we let p_2 resume and complete its call to setValidBlock$(b_0, b'_{2,1})$ which must also succeed and return $\top$ as our oracle has a branching factor of 2. By construction, p_2 sees the two branches in its following update_view() of bt (i.e., chain $b_0, b'_{1,h}$ with $h = 1$, and chain $b_0, b'_{2,1}$) of the same length thus the selection function f_a selects the branch $b_0, b'_{2,1}$ for where to append the next block as block $b'_{2,1}$ is smaller than $b'_{1,h}$ in the lexicographical order. We let p_2 append blocks to this branch until some time t_4 at which it becomes poised to call setValidBlock$(b'_{2,c}, b'_{2,c+1})$ with $c = 2$ such that the length of the chain $b_0, \ldots, b'_{2,c}$ is smaller than the chain $b_0, \ldots, b'_{1,h+1}$, or in case of equal length has a higher lexicographical order, and such that the length of the chain $b_0, \ldots, b'_{2,c+1}$ is greater than the chain $b_0, \ldots, b'_{1,h+1}$, or in case of equal length has a smaller lexicographical order.

As before, it is time to stop the execution of p_2 and resume the execution of p_1 and to let it complete its call to setValidBlock$(b'_{1,h}, b'_{1,h+1})$. We can continue to create two infinite branches sharing only the root by alternatively letting p_1 and p_2 extend their own branch while stopping one and resuming the execution of the other each time its length would overcome the length of the other branch extended with the pending block (and the appropriate lexicographical orderings in case of equal length). This way we construct a tree composed of two infinite branches sharing only the root b_0 as common prefix. It is easy to see that we can integrate read operations that may return the current chain from any branch as both branches are temporarily the longest one. Thus, the common prefix never increases, and so, the eventual finality consistency criteria is not satisfied.

It is important to note that with any length function that increases monotonically with prefixes (e.g, the length function could count the total number of transactions that belong to the blocks on the same branch) then this scenario still holds. In that case h and c in the proof could be larger than 1 and 2 respectively. ◀

4.2 Asynchronous Solution to $\mathcal{EF}^\star$ 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 $i \in \mathbb{N}$ and is equipped with signatures that can be used to identify the message sender identifier. Each block is identified with the identifier of the process that created it. Block identifier is inserted in the header of the block. Moreover, each process is equipped with an Eventual BFT-Reliable Broadcast primitive. If a correct process p broadcasts a message m then p eventually delivers m and if a correct process p delivers m then all correct processes eventually deliver m. We assume the system is such that we can implement an eventual reliable broadcast primitive, e.g., we assume that the infinite set of processes are arranged in a topology in which for each pair of correct processes, there exists a path composed by only correct processes [20]. Moreover, as proved in [1] reliable communications are necessary for eventual finality. We show that in that setting it is possible to build a blockchain that satisfies eventual prefix consistency.

The main idea of Algorithm 1 consists in using local selection functions f_a and f_r for append and read operations respectively and characterizing blocks by their parent and the producer signature. Let us first describe the append() and read() operations first and the selection function after.

To perform an append() operation of a block b, processes extend the chain returned by function f_a applied on their current view of bt with b, i.e., $f_a(bt)^\frown b$, and rb-broadcast

Algorithm 1 $\mathcal{EF}^\star$ with an unbounded number of Byzantine processes

```
1 upon rb-delivery(bc)
2 |  bt.addIfValid(bc)
3 end
4 upon append(b)
5 |  rb-broadcast(f_a(bt)⁀b)
6 end
7 upon read()
8 |  return f_r(bt)
9 end
```

$f_a(bt)^\frown b$. When a process rb-delivers a blockchain bc, it calls bt.addIfValid(bc) that merges bc with bt if the former is valid. By merging bc with bt we mean that for each block b_i of bc starting from the genesis block b_0, if b_i is not present in bt then b_i is added to bt, i.e., b_i is added to the block of bt whose hash is equal to the one contained in b_i's header. For read() operations, processes return the chain selected by f_r on their current bt.

Given a blocktree bt, the append selection function f_a selects a chain in bt by going from the root (i.e., genesis block) to a leaf, choosing at each fork b_i the edge to the child with the lowest identifier. If more than one child have the same identifier (i.e., they have been created by the same process), then all of them are ignored. If all the children have the same identifier, then f_a returns the chain from the genesis block to b_i. Blocks are ranked by the creator identifier, in the domain of the natural number and thus lower bounded by 0. Then even though, an infinite number of blocks is added continuously to a fork, there is not, for a given block, an infinite number of blocks with a smaller identifier. Thus eventually the selection function f_a will always select the same prefix. Finally, since blocks are diffused by a rb-broadcast primitive, eventually all correct processes will have the same view of the blocktree. When a process invokes the read() operation, it returns the blockchain selected by the read selection function f_r applied to its current view of the blocktree. By imposing that $f_r = f_a$, then eventually all the processes, when reading, will select the same prefix.

▶ **Theorem 14.** *Algorithm 1 is a solution for $\mathcal{EF}^\star$ in an asynchronous system with a possibly infinite set of processes which can append infinitely many blocks, and suffer from an unbounded number of Byzantine failures.*

Proof. We show by construction that Algorithm 1 solves $\mathcal{EF}^\star$ in an asynchronous system with a possibly infinite set of processes which can append infinitely many blocks, and can suffer an unbounded number of Byzantine failures. Intuitively, despite the unbounded number of blocks in each fork, by the eventual reliable broadcast, eventually for each fork all correct processes have the same block with the smallest identifier. Hence, by the read selection function that at each fork selects the block with the smallest identifier in order to select the chain to read, eventually, at all correct processes, function f_r returns the blockchain having a common increasing prefix. Let $p_1, p_2, \dots,$ be a possibly infinite set of processes, such that each one maintains its local view bt_i of blocktree bt by running Algorithm 1. Then for any correct process p_i the following properties hold.

- **Chain validity:** it is satisfied by function bt.addIfValid(bc) that merges blockchain bc to bt_i only if the former is valid.
- **Chain integrity:** The read() operation returns the chain of blocks selected by function f_r applied to bt_i. By Line 2 of Algorithm 1, only valid blocks are locally added to bt_i

once they have been reliably delivered. By Algorithm 1, the only place at which blocks are reliably broadcast is in the append() operation.

- **Eventual prefix:** The eventual prefix property follows from the definition of f_a and the reliable broadcast. Thanks to the reliable broadcast for any b in the bt of a correct process p, eventually all correct processes deliver b. Let t_b be the time after which no process can append further blocks b_{child} to b such that b_{child} is part of the chain returned by f_a. This time t_b always exists, as for each block b having potentially infinitely many children we have that, by definition of function f_a, $f_a(bt)$ selects a chain in bt by going from the root to a leaf, choosing at each fork b the edge to the child with the lowest identifier. Since identifiers are lower bounded by 0, eventually function f_a will always select the same child b' of b. The same argument applies for b' and its children. Hence, if any two correct processes invoke the read operation infinitely many times, then as $f_r = f_a$, eventually they return chains that satisfy the eventual prefix property.
- **Ever growing tree:** 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.

◀

4.3 Eventually Synchronous Solution to $\mathcal{EF}^{\Diamond c}$ with less than half of Byzantine Processes

In this section we prove that $\mathcal{EF}^{\Diamond c}$ is solvable in an eventual synchronous message-passing system with less than $n/2$ Byzantine processes, where n is the number of processes.

We propose an algorithm, called $\mathcal{AF}$ for Accountable Forking. This algorithm is inspired by the Streamlet [7] algorithm. Streamlet [7] assumes the presence of less than a third of Byzantine processes and an eventual synchronous system with a known message delay Δ after GST. We weaken both of these assumptions to provide a solution to $\mathcal{EF}^{\Diamond c}$ (or equivalently to the eventual immediate finality, see Theorem 10). 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 then present our protocol in terms of proposed modifications to Streamlet, before providing the proof.

Streamlet protocol. The Streamlet protocol works in an eventually 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. [1]

In Streamlet [7], each epoch, composed of 2Δ, has a designated leader chosen at random by a publicly known hash function. Each block b is labelled with the epoch ($b.epoch$) at which it has been created. This allows processes to establish if a block b has been created by a legitimate leader. The protocol works as follows:

- **Propose-Vote.** In every epoch:
  - The epoch's designated leader proposes a new block (rb-broadcast it, rb-broadcast as defined in Section 4.2) extending from the longest notarized chain (defined in a moment) it has seen, if there are multiple then it breaks ties arbitrarily.

[1] Notice that, in Streamlet [7] there is not the notion of time but of round, which denotes a basic unit of time.

- Every process votes (rb-broadcast a vote) 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.

- **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.

Our protocol $\mathcal{AF}$ is such that for any given fork, correct processes can blame the process that originates it, i.e, a Byzantine process creating a fork is accountable for it. $\mathcal{AF}$ makes the following two 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. 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, i.e, $b.epoch = b'.epoch$;
- **Cond. 2.** either $((b.epoch < b'.epoch)$ and $(b.height > b'.height))$ or $((b'.epoch < b.epoch)$ and $(b'.height > b.height))$. Function height counts the number of blocks from the genesis block.

If a process votes for blocks inconsistent with one another then it is detected as Byzantine. Once a correct process p detects a Byzantine process q, p ignores all messages coming from q. Since all messages received by a correct process q are received by any correct process, then all of them do the same with respect to q.

▶ **Theorem 15.** *There exists a solution that solves unknown bounded revocation eventual finality in an eventual synchronous system with less than $n/2$ Byzantine processes, where n is the number of processes participating to the algorithm.*

Proof. Let us show that $\mathcal{AF}$ is a solution for unknown bounded revocation eventual finality.

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. Hence 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' belonging to the prefix of b'_3 such that $b'.epoch > b_1.epoch$ and $b'.height$ is the smallest in the prefix of b'_3. Such block always exists as b'_1 satisfies those two conditions. We have two cases: Either $b'.height < b_3.height$ or $b'.height \geq b_3.height$. In the former case, b' is inconsistent

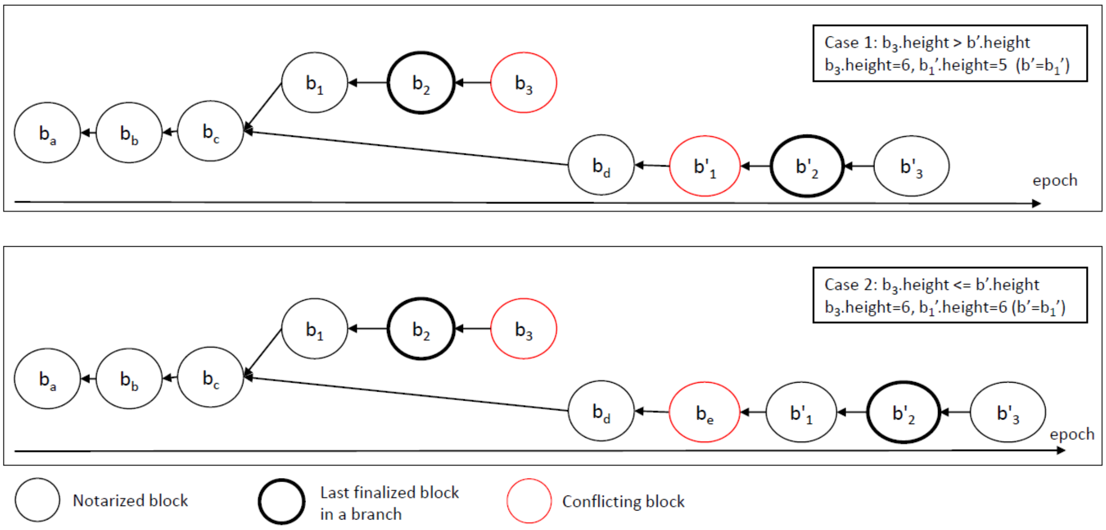

Figure 2 Illustration of block inconsistencies due to the occurrence of a fork when the finalized blocks are not labelled with the same epoch. Epochs are on the x axis, and all consecutive blocks have consecutive epochs, e.g., b_c and b_d have four epochs of difference, 4 and 7 respectively, while b_1 and b_2 are labelled with consecutive epochs.

with b_3 since by assumption $b'.epoch > b_3.epoch$. In the latter case, the predecessor of b' is inconsistent with b_3. Indeed, the predecessor of b' has a strictly smaller height than b_1 and by assumption has a larger epoch number than b_3. Figure 2 illustrates the presence of inconsistent blocks in presence of a fork at some block b_c. From b_c two chains are built, the first one consisting of the sequence of three blocks b_1, b_2 and b_3, and the second chain consisting of four consecutive blocks b_d, b'_1, b'_2, b'_3 (to illustrate the first case) and of five consecutive blocks $b_d, b_e, b'_1, b'_2, b'_3$ (to illustrate the second case). In both cases block block b'_1 plays the role of block b'. In the first case (figure in the top), $b_3.height = 6$ and $b'.height = 5$ while $b_3.epoch = 6$ and $b'.height = 5$. Thus Cond. 2 applies. In the second case (figure in the bottom), since $b'.height \geq b3.height$ then there must exist some block b_e in the b' prefix. Thus $b_e.height < b'.height$. Moreover, given that by assumption $b_e.epoch > b_3.epoch$, then Cond. 2 holds for b_e and b_3.

Hence there is always a couple of inconsistent blocks in a fork.

Let us now conclude our proof that we solve the eventual immediate finality. If a fork occurs, then each correct process eventually detects at least one Byzantine process and ignores 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, our protocol $\mathcal{S}$ remains live as in the original Streamlet protocol. $\mathcal{S}$ also inherits the properties of the original Streamlet protocol for finalizing blocks eventually when synchrony is reached. ◀

5 Conclusion

In this work we have focused on the formalisation of eventual finality, which ensures that selected main chains at different processes share a common increasing prefix. We have formalised different forms of eventual finality in terms of the maximal number of blocks that can be revoked at each reconciliation, which is a crux in current blockchain designs.

We have formally shown that in an asynchronous system is not possible to reach a bound on the number of blocks that can be revoked. On the other hand, we proposed for the first time a solution in an eventually synchronous system with less than half of Byzantine processes guaranteeing that such bound is reached eventually. We have also shown that in an asynchronous system eventual finality with no bound on the number of revocable blocks cannot be solved using the reconciliation rule of Bitcoin. Still we provide an asynchronous solution with an unlimited number of Byzantine processes. We hope that these results will better guide blockchain designs and link them to clear formal properties of finality.

References

1. Anceaume, E., Pozzo, A. D., Ludinard, R., Potop-Butucaru, M., and Tucci Piergiovanni, S. Blockchain abstract data type. In Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA) (2019).
2. Androulaki, E., and et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In Proceedings of the European Conference on Computer Systems (EuroSys) (2018).
3. Anta, A. F., Konwar, K., Georgiou, C., and Nicolaou, N. Formalizing and implementing distributed ledger objects. ACM SIGACT News 49, 2 (2018), 58–76.
4. Aştefanoaei, L., Chambart, P., Pozzo, A. D., Rieutord, T., Tucci-Piergiovanni, S., and Zălinescu, E. Tenderbake - a solution to dynamic repeated consensus for blockchains. In Proceedings of the Fourth International Symposium of Foundations and Applications of Blockchain (2021).
5. Buterin, V., and Griffith, V. Casper the friendly finality gadget. CoRR (2017).
6. Cachin, C. Blockchain - From the Anarchy of Cryptocurrencies to the Enterprise (Keynote Abstract). In Proceedings of the International Conference on Principles of Distributed Systems (OPODIS) (2016).
7. Chan, B. Y., and Shi, E. Streamlet: Textbook streamlined blockchains. `https://eprint.iacr.org/2020/088.pdf`, 2020.
8. Chen, J., and Micali, S. Algorand: A secure and efficient distributed ledger. Theor. Comput. Sci. (2019).
9. Crain, T., Gramoli, V., Larrea, M., and Raynal, M. (leader/randomization/signature)-free byzantine consensus for consortium blockchains. CoRR abs/1702.03068 (2017).
10. Dubois, S., Guerraoui, R., Kuznetsov, P., Petit, F., and Sens, P. The weakest failure detector for eventual consistency. In Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC) (2015).
11. Garay, J. A., Kiayias, A., and Leonardos, N. The bitcoin backbone protocol: Analysis and applications. In Proc. EUROCRYPT International Conference (2015).
12. Goodman, L. Tezos – a self-amending crypto-ledger, 2014.
13. Grigg, I. EOS: An introduction. `https://whitepaperdatabase.com/eos-whitepaper/`.
14. Guerraoui, R., Kuznetsov, P., Monti, M., Pavlovič, M., and Seredinschi, D.-A. The consensus number of a cryptocurrency. In Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing (PODC) (2019).
15. Herlihy, M. Blockchains and the future of distributed computing. In Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC) (2017).
16. Kiayias, A., Russell, A., David, B., and Oliynykov, R. Ouroboros: A provably secure proof-of-stake blockchain protocol. In Proceedings of the Advances in Cryptology (2017).
17. Koltsov, A., Cheremensky, V., and Kapulkin, S. Casper White Paper.
18. Lamport, L., Shostak, R., and Pease, M. The Byzantine generals problem. ACM Transactions on Programming Languages and Systems (1982).
19. Liskov, B., and Zilles, S. Programming with abstract data types. ACM SIGLAN Notices 9, 4 (1974).

20 Maurer, A., and Tixeuil, S. On byzantine broadcast in loosely connected networks. In Proceedings of the 26th International Symposium on Distributed Computing (DISC) (2012).
21 Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system. www.bitcoin.org (2008).
22 Nomadic Labs. Analysis of Emmy$^+$. https://blog.nomadic-labs.com/analysis-of-emmy.html, 2019.
23 Pass, R., Seeman, L., and Shelat, A. Analysis of the blockchain protocol in asynchronous networks. In Proceedings of the EUROCRYPT International Conference (2017).
24 Perrin, M. Distributed Systems, Concurrency and Consistency. ISTE Press, Elsevier, 2017.
25 Raynal, M. Eventual leader service in unreliable asynchronous systems: Why? how? In Proceedings of the IEEE International Symposium on Network Computing and Applications (NCA) (2007).
26 Stewart, A. Poster: Grandpa finality gadget. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (2019), CCS '19, p. 2649–2651.
27 Wood, G. Ethereum: A secure decentralised generalised transaction ledger. http://gavwood.com/Paper.pdf.