On the Synchronization Power of Token Smart Contracts

Orestis Alpos
University of Bern
orestis.alpos@inf.unibe.ch

Christian Cachin
University of Bern
cachin@inf.unibe.ch

Giorgia Azzurra Marson
University of Bern
giorgia.marson@inf.unibe.ch

Luca Zanolini
University of Bern
luca.zanolini@inf.unibe.ch

Abstract

Modern blockchains support a variety of distributed applications beyond cryptocurrencies, including *smart contracts*, which let users execute arbitrary code in a distributed and decentralized fashion. Regardless of their intended application, blockchain platforms implicitly assume consensus for the correct execution of a smart contract, thus requiring that all transactions are totally ordered. It was only recently recognized that consensus is not necessary to prevent double-spending in a cryptocurrency (Guerraoui *et al.*, PODC'19), contrary to common belief. This result suggests that current implementations may be sacrificing efficiency and scalability because they synchronize transactions much more tightly than actually needed.

In this work, we study the synchronization requirements of Ethereum's ERC20 token contract, one of the most widely adopted smart contacts. Namely, we model a smart-contract token as a concurrent object and analyze its consensus number as a measure of synchronization power. We show that the richer set of methods supported by ERC20 tokens, compared to standard cryptocurrencies, results in strictly stronger synchronization requirements. More surprisingly, the synchronization power of ERC20 tokens depends on the object's state and can thus be modified by method invocations. To prove this result, we develop a dedicated framework to express how the object's state affects the needed synchronization level. Our findings indicate that ERC20 tokens, as well as other token standards, are more powerful and versatile than plain cryptocurrencies, and are subject to *dynamic* requirements. Developing specific synchronization protocols that exploit these dynamic requirements will pave the way towards more robust and scalable blockchain platforms.

1 Introduction

The rise of cryptocurrencies has motivated the development of distributed applications running over blockchain platforms. These applications go far beyond the concept of a decentralized cryptocurrency, as initially envisioned by Bitcoin [21]. Taking this diversity to the extreme, *smart contracts* enable a blockchain to execute arbitrary programs, in a fully decentralized fashion akin to a "world computer." Introduced by Ethereum [11], smart contracts come in many different flavors and are the key element in most blockchain projects today.

Regardless of the type of supported smart contract, blockchain platforms rely on a distributed protocol that orders transactions and emulates a ledger data structure. A transaction may be a simple "coin transfer" in a cryptocurrency or a complex method call to a decentralized application. For either use-case, it is widely accepted that the blockchain nodes must execute all transactions in the same order to ensure consistency [26, 24]. That is, to ensure that the emulated ledger is consistent, transactions are sent using protocols that implement *total-order broadcast* or *consensus*. Garay *et al.* [14] showed such an equivalence formally for the Bitcoin protocol. This common theme seems to suggest that total order is also *necessary* for the consistency of blockchains.

However, this folklore intuition is wrong: Recent work by Guerraoui *et al.* [16] shows that consensus is not necessary to avoid double-spending in cryptocurrency applications. After distilling the essence of

a cryptocurrency protocol to the problem of realizing a consistent *asset transfer* (AT), the authors cast the latter as a sequential object in the shared-memory model and prove the AT object has *consensus number* 1 in the wait-free hierarchy [18]. In other words, consensus is not needed at all for emulating the functions of Bitcoin! The consensus number is a well-established tool to express the synchronization requirements of asynchronous concurrent objects. Informally, it provides an upper bound for the number of processes that can be synchronized using (arbitrarily many) instances of a given object. For cryptocurrencies modeled after Bitcoin that support shared accounts with up to k owners, Guerraoui *et al.* [16] introduce a k-*shared* asset transfer (k-AT) object that has consensus number k, which is as powerful as consensus among its k owners. Going beyond their theoretical elegance, these results are of great practical interest because they pave the way to consensus-free implementations of cryptocurrencies [6, 17], with higher efficiency and robustness to network partitions. In this particular case, for example, only the k owners need to reach consensus for spending from the account, provided they have additional means to publicize this widely in the network.

In this work, we investigate the synchronization power of smart contracts. We observe that although k-AT would allow to *generically* implement any smart contract among k processes, it remains open whether this level of synchronization is *necessary* for widely-used blockchain applications. We focus our attention on smart contracts for Ethereum, which is by far the most important platform for hosting decentralized applications. Moreover, many other networks have adopted its programming model. We present an abstraction of a token object that captures and generalizes the functionality of an ERC20 contract [28], which forms the basis for countless applications on Ethereum that hold billions today. Notice that the k-AT abstraction [16] applies to Bitcoin and its UTXO model of a currency. Ethereum, on the other hand, uses accounts, and ERC20 contracts are considerably more powerful than Bitcoin transactions. The additional features of ERC20 make it possible, for example, to let account owners *conditionally* issue transfers to other users of their choosing.

Empowering account owners to approve other spenders makes the ERC20 token object strictly more powerful than k-AT. In addition, approval of new spenders can be performed flexibly, at any time and for arbitrary amounts of tokens, achieving a dynamic that has no counterpart in the case of k-AT. Because of these differences, the results established for k-AT [16] cannot be lifted to ERC20 tokens. What crucially distinguishes an ERC20 token object from k-shared asset transfer is the increased level of dynamicity, which is reflected in its synchronization requirements. Namely, the consensus number of ERC20 tokens depends on the number of approved spenders for the same account, which may change as the account owner enables more spenders. Based on the observations, we develop a formalism to express that the consensus number of a token object can change over time, depending on the object's state. More concretely, we prove that there exist specific states from which it is possible to solve consensus among k processes, for every $k \leq n$ where n is the number of accounts defined by the token contract. Moreover, these states can be reached by letting any of the account owners approve new spenders.

Establishing the synchronization power of smart contracts is important for understanding the level of synchronization that is required to run decentralized applications in a blockchain network. Not every two users must be synchronized on every aspect of their respective states, this only matters when their actions affect each other. Identifying the level of consensus needed for different applications also paves the way for realizing more efficient blockchain networks, which may exploit more parallelism.

Organization. After discussing related work in Section 2, in Section 3 we present relevant notations and background concepts. We then describe ERC20 tokens in Section 4, and we analyze their synchronization requirements in Section 5. In Section 6, we discuss other notable token standards and elaborate on extending our results to these tokens. Section 7 concludes our work suggesting future research directions.

2 Related Work

Synchronization requirements and blockchain scalability. Guerraoui *et al.* [16] observe that consensus is not necessary to realize a decentralized cryptocurrency. They propose a shared-memory abstraction for the asset transfer problem as implemented in Bitcoin [21] and show that it requires only a minimal level of synchronization. Specifically, they show that asset transfer has consensus number 1 in Herlihy's wait-free hierarchy [18]. The approach of analyzing the synchronization requirements of shared objects in terms of consensus number has been used by others. For instance, Cachin *et al.* [5] study the consensus number of various cloud-storage abstractions, and find that a key-value store has the weakest synchronization power (i.e., its consensus number is 1) while a replica object requires the strongest synchronization level (i.e., its consensus number is ∞).

Obviating the need to reach agreement on the exact ordering of transactions opens the door to more scalable solutions than the currently deployed, consensus-based blockchains. In this context, Collins *et al.* [6] present a decentralized payment system based on Byzantine reliable broadcast. Guerraoui *et al.* [17] generalize the Byzantine reliable broadcast abstraction to the probabilistic setting and propose a protocol which efficiently realizes it, with the goal of replacing the usual quorum-based safety notions with stochastic guarantees for consistency in a distributed network. While the above-mentioned protocols fulfill the synchronization requirements for implementing asset transfer, and can therefore support plain cryptocurrency applications, they are not sufficient for the implementation of generic smart contracts.

Many other approaches have been explored in order to increase blockchain scalability [7], most prominently "on-chain" proposals such as optimized BFT-based consensus protocols [15, 3, 30], DAG-based protocols [19, 25], and sharding [31, 20, 29, 12], as well as "off-chain" solutions such as payment channels [22, 9] and sidechains [2]. Even though these alternative approaches have received a lot of attention recently [32], they have not yet been widely adopted in practice.

Smart contracts and Ethereum tokens. Ethereum [11] is the first open-source cryptocurrency platform supporting smart contracts, providing a decentralized virtual machine for executing arbitrary Turing-complete programs. The ERC20 standard, introduced by Buterin and Vogelsteller [28], provides functions for handling tokens over Ethereum, allowing users to transfer various types of transferable goods such as digital and physical assets. It formulates a common interface for fungible tokens and has become the most widely-deployed API for implementing a token functionality, with more than half of the overall Ethereum transactions being ERC20 token transfers [27].

3 Preliminaries

3.1 Shared Memory and Synchronization Power of Shared Objects

We begin with presenting well-established concepts from the concurrent computing literature. We mostly follow the standard notations and nomenclature [4, 23, 16].

Concurrent objects. We assume a (finite) set Π of processes that communicate in an asynchronous manner by invoking operations on, and receiving responses from, shared objects. Processes are sequential, meaning that no process invokes a new operation before completing (i.e., receiving the response from) all previously invoked operations. We assume a *crash-failure* model: a process may halt prematurely, in which case we say the process has crashed. We say that a process is *faulty* if it crashes during its execution, otherwise we say that it is *correct*.

An *object type* (or simply *object*) defines the functionality of shared-memory programming abstractions providing a finite set of operations. We consider *concurrent* objects, namely objects which can be accessed by multiple processes simultaneously and concurrently. The specification of these objects can be sequential or not, where "sequential" means that all correct behaviors of the object can be described

with sequences of invocations and responses (traces). In this paper, we are only concerned with sequential objects. We define an object type as a tuple $T = (Q, q_0, O, R, \Delta)$, where Q is a set of states, $q_0 \in Q$ is an initial state, O is a set of operations, R is a set of responses, and $\Delta \subseteq Q \times \Pi \times O \times Q \times R$ defines the valid state transitions. We write $(q, p, o, q', r) \in \Delta$ to denote that process p invokes operation o on the object in current state q, and the operation completes by returning response r and causing the object to enter state q'.

An *implementation* for an object type T is a distributed algorithm describing, for each process, sufficient steps to realize each of the object's operations in such a way that desired safety and liveness properties are met. The strongest liveness condition for object implementations is *wait-freedom* [18], requiring that every invocation of any object operation terminates, despite process failures.

Registers. The simplest object type is a *register*, which defines a shared-memory functionality providing *read* and *write* operations. Given a register R, a process can write a value v into R by invoking $R.write(v)$; upon completion of this operation, the process is given TRUE in response. Similarly, a process can initiate a read operation on R by invoking $R.read()$; the process obtains a value R stores. In the paper, we consider *atomic* registers. Formally, an atomic register provides *termination*, i.e., if a correct process invokes an operation, then the operation eventually completes, and *validity*, i.e., a read that is not concurrent with a write returns the last value written, while a read that is concurrent with a write returns the last value written or the value concurrently being written. Moreover, an atomic register provides *ordering*, i.e., if a read returns a value v and a subsequent read returns a value w, then the write of w does not precede the write of v. This property implies that every operation of an atomic register can be thought to occur at a single indivisible point in time, which lies between the invocation and the completion of the operation [4].

Consensus. Another important object type is *consensus*, which allows a set of processes to agree on a value. A consensus object C provides a single operation *propose*. A process can invoke $C.propose(v)$ on input a proposal v as a candidate value to be agreed upon. Every process can call *propose* with their own proposed value, and only one invocation is permitted (i.e., it is a "single-shot" object). Upon completion, the operation returns a value d, called the decided value. Besides wait-freedom (a.k.a. *termination*), we require *validity*, i.e., the decided value is the proposal v of some process, and *consistency*, i.e., every process returns the same decided value d.

Asset transfer (AT). The asset transfer object was proposed by Guerraoui *et al.* [16] as an abstraction for cryptocurrencies. Let $\mathcal{A}$ be a finite set of accounts, $|\mathcal{A}| = n$, and let $\mu \colon \mathcal{A} \to 2^{\Pi}$ denote the owner map that associates each account $a \in \mathcal{A}$ to the set of processes sharing the account. If $|\mu(a)| = k$, we say that a is a k-shared account.

Definition 1 (Asset transfer). The *asset transfer* object associated to $\mathcal{A}$ and μ, denoted by $AT = (Q, q_0, O, R, \Delta)$, is defined as follows:

- Set Q contains all *balance* maps, i.e.,

$$Q = \{\beta \colon \mathcal{A} \to \mathbb{N}\}. \tag{1}$$

- The initialization map $q_0 = \beta_0$ assigns an initial balance to each account.
- O contains two operations, $O = \{\textit{transfer}(a_s, a_d, v) : a_s, a_d \in \mathcal{A}, v \in \mathbb{N}\} \cup \{\textit{balanceOf}(a) : a \in \mathcal{A}\}$, where $\textit{transfer}(a_s, a_d, v)$ lets the caller process, say p, transfer v tokens from a source account a_s to a destination account a_d, provided that $p \in \mu(a_s)$, and $\textit{balanceOf}(a)$ reads the balance of account a.
- R contains the possible responses to operations in O, $R = \{\text{TRUE}, \text{FALSE}\} \cup \mathbb{N}$.

- Δ defines the valid state transitions. Given a state $q = \beta \in Q$, a process $p \in \Pi$ with account a_p, an operation $o \in O$, a response $r \in R$, and a new state $q' = \beta' \in Q$, we have $(q, p, o, r, q') \in \Delta$ if and only if either of the following conditions holds:
  - $o = \textit{transfer}(a_s, a_d, v) \ \wedge \ p \in \mu(a_s) \ \wedge \ \beta(a_s) \geq v \ \wedge \ \beta'(a_s) = \beta(a_s) - v \ \wedge \ \beta'(a_d) = \beta(a_d) + v \ \wedge \ \forall c \in \mathcal{A} \setminus \{a_s, a_d\} : \beta'(c) = \beta(c) \ \wedge \ r = \text{TRUE}$;
  - $o = \textit{transfer}(a_s, a_d, v) \ \wedge \ (\beta(a_s) < v \ \vee \ p \notin \mu(a_s)) \ \wedge \ q' = q \ \wedge \ r = \text{FALSE}$;
  - $o = \textit{balanceOf}(a) \ \wedge \ q' = q \ \wedge \ r = \beta(a)$.

If the maximum number of processes sharing an account is k, we name the object a k-shared asset transfer, and we denote it by k-AT.

Synchronization power of shared objects. The prominent result by Fischer, Lynch, and Paterson [13] establishes the impossibility of wait-free implementing consensus from atomic registers. This means that consensus requires a higher level of synchronization than atomic registers. In fact, the consensus object is *universal*, in the sense that any shared object described by a sequential specification can be wait-free implemented from consensus objects and atomic registers [18]. Therefore, consensus can be used to reason about the synchronization power of all shared objects (which admit a sequential specification) among a number of processes. This leads to the central concept of *consensus number* to express the synchronization power of shared objects.

Definition 2 (Consensus number [18]). The *consensus number* associated with an object O is the largest number n such that it is possible to wait-free implement a consensus object from atomic registers and objects of type O, in a system of n processes. If there is no largest n, the consensus number is said to be infinite. Given an object O, we denote its consensus number by $\mathcal{CN}(O)$.

The consensus number allows comparing objects based on their synchronization power, thereby establishing a hierarchy among objects—the consensus hierarchy. In this work, we leverage the concept of consensus number to study the level of synchronization required for popular smart-contracts tokens.

Theorem 1 ([18]). *Let O and O' be two objects such that $\mathcal{CN}(O) = n$ and $\mathcal{CN}(O') > \mathcal{CN}(O)$. Then there is no wait-free implementation of an object of type O' from objects of type O and read/write registers in a system of n processes.*

4 Defining ERC20 Tokens as Shared Objects

In this section, we present a smart contract for transferring tokens defined, by the Ethereum Request for Comment (ERC) 20 specification, and propose a corresponding shared-memory abstraction.

Tokens are blockchain-based assets which can be exchanged across users of a blockchain platform. Ethereum Request for Comment (ERC) 20 defines a standard for the creation of a specific type, dubbed ERC20 token, one of the most widely adopted tokens on Ethereum. ERC20 tokens are transferred through dedicated transactions among Ethereum addresses, and are managed by smart contracts. For completeness, we reproduce in Appendix A the algorithmic specification defined in the EIP-20 proposal [28], with minimal notational changes to ease the comparison with the objects defined in this paper.

The definition of a token object we propose is a generalization of an ERC20 token. The reason for slightly deviating from the original specification (as per Algorithm 3, Appendix A) is that it represents a more expressive abstraction: it allows us to reason about synchronization requirements of ERC20 tokens as well as comparing them with the asset transfer object.

Let $\mathcal{A}$ be a finite set of accounts. We assume one account per process, $|\Pi| = |\mathcal{A}| = n$, and define a bijection $\omega \colon \mathcal{A} \to \Pi$ between accounts and processes, i.e., $\omega(a_i) = p_i$ for all $i \in \{1, \ldots, n\}$. We name $\omega \colon \mathcal{A} \to \Pi$ the *owner map* that associates to each account a the corresponding process $\omega(a)$ which

owns the account.[1] To simplify the notation, we use the shorthand a_p for the account owned by process p, i.e., such that $\omega(a_p) = p$.

Notice that in the case of asset transfer (cf. Definition 1), *account ownership* captures a slightly different setting than compared to token objects: the former allows for shared ownership while the latter does not. We make this explicit by using different owner maps μ and ω, respectively. However, as we explain in detail in the next section, ERC20 tokens offer a richer set of operations that, among others, enables a conditioned form of shared ownership.

Using this notation, below we provide a specification for the *ERC20 token* smart contract using the formalism of shared objects (cf. Section 3.1).

Definition 3 (ERC20 token object). Let $\mathcal{A}$ be a set of accounts and let Π be the set of corresponding owner processes. A *token object* T associated to $\mathcal{A}$ consists of a tuple $T = (Q, q_0, O, R, \Delta)$, where:

States: Q contains all *balances* maps and *allowances* maps, i.e.,

$$Q = \{\beta \colon \mathcal{A} \to \mathbb{N}\} \times \{\alpha \colon \mathcal{A} \times \Pi \to \mathbb{N}\}. \tag{2}$$

Intuitively, for $a \in \mathcal{A}$ and $p \in \Pi$, $\beta(a)$ indicates the balance of account a, and $\alpha(a, p)$ denotes the amount of tokens that process p is allowed to spend from account a.

Initial state: $q_0 = (\beta_0, \alpha_0)$ denotes the pair of initial account balances and allowances.

Operations: O contains the following operations:

$$O = \{\textit{transfer}(a_d, v) : a_d \in \mathcal{A}, v \in \mathbb{N}\} \tag{3}$$
$$\cup \{\textit{transferFrom}(a_s, a_d, v) : a_s, a_d \in \mathcal{A}, v \in \mathbb{N}\} \tag{4}$$
$$\cup \{\textit{approve}(p, v) : p \in \Pi, v \in \mathbb{N}\} \tag{5}$$
$$\cup \{\textit{balanceOf}(a) : a \in \mathcal{A}\} \tag{6}$$
$$\cup \{\textit{allowances}(a, p) : a \in \mathcal{A}, p \in \Pi\}. \tag{7}$$

Operation *transfer*(a_d, v) lets the caller process, say p, transfer v tokens from its account a_p to destination account a_d; similarly, *transferFrom*(a_s, a_d, v) lets the caller process transfer v tokens from source account a_s to destination account a_d. Operation *approve*(p', v) lets the caller process p authorize another process p' to transfer up to v tokens from p's account. Finally, *balanceOf*(a) reads the balance of account a, and *allowances*(a, p) reads the amount of tokens that process p is authorized to transfer from a.

Responses: R contains the possible responses for all operations in O, namely $R = \{\text{TRUE}, \text{FALSE}\} \cup \mathbb{N}$.

Sequential specification: Δ defines the valid state transitions. Given a state $q = (\beta, \alpha) \in Q$, a process $p \in \Pi$ with account a_p, an operation $o \in O$, a response $r \in R$, and a new state $q' = (\beta', \alpha') \in Q$, we have $(q, p, o, r, q') \in \Delta$ if and only if either of the following conditions holds:

- $o = \textit{transfer}(a_d, v) \ \wedge \ \beta(a_p) \geq v \ \wedge \ \beta'(a_p) = \beta(a_p) - v \ \wedge \ \beta'(a_d) = \beta(a_d) + v \ \wedge \ \forall c \in \mathcal{A} \setminus \{a_p, a_d\} : \beta'(c) = \beta(c) \ \wedge \ \alpha' \equiv \alpha \ \wedge \ r = \text{TRUE}$;
- $o = \textit{transfer}(a_d, v) \ \wedge \ \beta(a_p) < v \ \wedge \ q' = q \ \wedge \ r = \text{FALSE}$.
- $o = \textit{approve}(\bar{p}, v) \ \wedge \ \alpha'(a_p, \bar{p}) = v \ \wedge \ \alpha'(a, p) = \alpha(a, p) \ \forall (a, p) \neq (a_p, \bar{p}) \ \wedge \ \beta' \equiv \beta \ \wedge \ r = \text{TRUE}$;
- $o = \textit{transferFrom}(a_s, a_d, v) \ \wedge \ \beta(a_s) \geq v \ \wedge \ \alpha(a_s, p) \geq v \ \wedge \ \beta'(a_s) = \beta(a_s) - v \ \wedge \ \alpha'(a_s, p) = \alpha(a_s, p) - v \ \wedge \ \beta'(a_d) = \beta(a_d) + v \ \wedge \ \alpha'(a, p) = \alpha(a, p) \ \forall (a, p) \neq (a_s, p) \ \wedge \ \forall c \in \mathcal{A} \setminus \{a_s, a_d\} : \beta'(c) = \beta(c) \ \wedge \ r = \text{TRUE}$;

[1] Although we use a similar formalism as Guerraoui *et al.* [16] to define account ownership, we make the restriction to single-owner accounts to meet the Ethereum-token specification. As we will see in later sections, in Ethereum tokens there are no shared accounts, however, a similar concept is enabled by means of dedicated methods.

- $o = \textit{transferFrom}(a_s, a_d, v) \ \wedge \ (\beta(a_s) < v \vee \alpha(a_s, p) < v) \ \wedge \ q' = q \ \wedge \ r = \text{FALSE}$;
- $o = \textit{balanceOf}(a) \ \wedge \ q = q' \ \wedge \ r = \beta(a)$;
- $o = \textit{totalSupply} \ \wedge \ q = q' \ \wedge \ r = \sum_{a \in \mathcal{A}} \beta(a)$;
- $o = \textit{allowances}(a, \bar{p}) \ \wedge \ q' = q \ \wedge \ r = \alpha(a, \bar{p})$.

The example below illustrates the various ERC20 token operations and their interplay.

Example 1 (ERC20 token: sample execution). Consider a set of three processes, $\Pi = \{A, B, C\}$ (Alice, Bob, and Charlie), and corresponding accounts, $\mathcal{A} = \{a_A, a_B, a_C\}$. Let Alice be the deployer of an ERC20 token contract, and suppose Alice provides an initial supply of 10 tokens, i.e., $\textit{totalSupply} = 10$. According to the ERC20 specification, the token object T associated to the contract is initialized as follows:

$$q_0 : \quad \textit{balances}[a_A, a_B, a_C] = [10, 0, 0], \quad \text{and}$$
$$\forall a \in \mathcal{A} : \textit{allowances}[a][A, B, C] = [0, 0, 0].$$

Starting from this initial configuration, let Alice invoke $\textit{transfer}(a_B, 3)$, sending 3 tokens to Bob's account. This operation triggers the transfer of 3 tokens from account a_A to account a_B and, upon completion, it causes the following state update:

$$q_1 : \quad \textit{balances}[a_A, a_B, a_C] \leftarrow [10 - 3, +3, 0].$$

Let now Bob invoke $\textit{approve}(C, 5)$, authorizing Charlie to transfer up to 5 tokens from account a_B. Upon completion, this operation causes the following state update:

$$q_2 : \quad \textit{allowances}[a_B] \leftarrow [0, 0, +5].$$

Upon being approved, let Charlie invoke $\textit{transferFrom}(a_B, a_C, 5)$ to transfer 5 tokens from Bob's account to his own account. Despite the fact that Charlie's allowance $\textit{allowances}[a_B][C] = 5$ would in principle permit such transfer, Bob's balance $\textit{balances}[a_B] = 3$ is currently insufficient. Therefore, the operation returns FALSE, leaving the state unmodified:

$$q_3 \leftarrow q_2$$

Finally, let Charlie invoke $\textit{transferFrom}(a_B, a_A, 1)$ to transfer 1 token from account a_B to Alice's account. This time, the amount of tokens to be transferred is below the account balance and, upon completion, the operation triggers the following state update:

$$q_4 : \quad \textit{balances}[a_A, a_B, a_C] \leftarrow [7 + 1, 3 - 1, 0] \quad \text{and} \quad \textit{allowances}[a_B] \leftarrow [0, 0, 5 - 1].$$

Further notation. In later sections of the paper, we will make use of the following shortcut notation. For every state $q \in Q$, we write T_q to denote the token object initialized with state q, i.e., $T = (Q, q, O, R, \Delta)$. Similarly, for $Q' \subseteq Q$ we write $T_{Q'}$ to indicate that token object is initialized with any state $q \in Q'$. We will also rely on an auxiliary token object $T|_{Q'}$, which is obtained from T by restricting the valid state transitions to remain within Q', i.e., $T|_{Q'} = (Q', q_0, O, R, \Delta')$ where $q_0 \in Q'$ and $\Delta' = \{(q, p, o, r, q') \in \Delta : q' \in Q'\}$. Finally, we note that in the ERC20 standard (cf. Algorithm 3, Appendix A) the state of the smart contract is fully specified by the arrays $\textit{balances}[\,]$ and $\textit{allowances}[\,]$. Namely, for all $a \in \mathcal{A}$ and all $p \in \Pi$, we have $T.\textit{balances}[a] = T.\beta(a)$ and $T.\alpha(a, p) = T.\textit{allowances}[a][p]$.

5 Consensus Number of ERC20 Tokens

In this section, we study the synchronization power of the ERC20 token object by analyzing its consensus number.

5.1 Overview of the Results

The consensus number of an ERC20 token object can be expressed in terms of the maximum number of processes that can transfer tokens from the same account. This number, denoted below by k, depends on the account balances and allowances defined by the object's state $q = (\beta, \alpha)$, and hence it can change as the state is updated. In the rest of this section, we therefore analyze the consensus number of a token object T for various state configurations.

Approach and challenges. Given the similarities between the ERC20 token object and the k-shared asset transfer object, one may think they have the same consensus number. Intuitively, the *approve* method in ERC20 tokens allows emulating shared accounts by letting every account owner authorize other processes to transfer tokens from its own account. In fact, there are at least two peculiarities of ERC20 tokens which depart from k-shared accounts. Firstly, a k-shared asset transfer supports at most k owners per account, where $k \leq n$ is fixed upfront (because the owner map μ in k-AT is *static*). This is in contrast with ERC20 tokens, where each account owner can *dynamically* add and remove spenders at any time of the execution, and the number of valid spenders per account is subject to change as the protocol is ongoing. In other words, an ERC20 token object could be loosely seen as a k-shared asset transfer with k changing dynamically. Secondly, in k-AT the owners of a shared account remain owners for the whole lifetime of the object, i.e., they all can transfer tokens from that account as long as the balance is positive. In ERC20 tokens instead, an approved spender remains a valid spender until it consumes the granted allowance or the account owner decides to revoke the spender's allowance (this can be done by resetting the allowance to the default value 0).

These crucial differences show a separation between the k-AT object and the ERC20 token object, and suggest that the two objects meet different synchronization requirements. In particular, it is not possible to apply known results and techniques for k-AT to the case of ERC20 tokens. Moreover, the approval mechanism to add and remove spenders in ERC20 tokens has subtle implications on the object's synchronization power.

In the rest of this section, we confirm these observations formally and make precise statements about the consensus number of the ERC20 token object. We now provide a rather informal summary of our results, which we state in full detail and prove in Section 5.2. The statements below hold for every $k \leq n$.

Lower bound. There exists a set S_k of states, which we name *synchronization states*, such that for every $q \in S_k$ it is possible to wait-free implement a consensus object among k processes using objects of type T_q (Theorem 2). Formally:

$$\mathcal{CN}(T_{S_k}) \geq k. \tag{8}$$

To prove this lower bound, we show that a consensus object supporting k processes reduces to T_q, with $q \in S_k$, by presenting a wait-free implementation of consensus for k processes from objects of type T_q and atomic registers.

Upper bound. The set of states can be partitioned into $[Q_1, \ldots, Q_n]$, with $Q = \cup_{k=1}^{n} Q_k$, so that for every $q \in Q_k$, at most k processes can reach consensus using token objects of type T_q (Theorem 3). Formally:

$$\mathcal{CN}(T_{Q_k}) \leq k. \tag{9}$$

Proving the upper bound turns out to be more involved. We proceed with an indirect argument, showing that the hypothesis $\mathcal{CN}(T_{Q_k}) = k' > k$ leads to a contradiction. Intuitively, the contradiction is reached because no implementation of consensus for k' processes from T_q, with $q \in Q_k$, can be wait-free.

5.2 Technical Results and Proofs

Essentially, we show that an ERC20 token represents a dynamic k-shared AT object, where k depends on the current object's state. Specifically, k is the maximum number of valid spenders for the same account.

For each $k \leq n$, where n is the total number of accounts, there exists a class S_k of states, the class of *k-synchronization states*, such that $\forall q \in S_k$, it holds $\mathcal{CN}(T_q) \geq k$. However, we cannot conclude that $\mathcal{CN}(T) = \infty$. We can only say that if a state $q \in S_n$ is reached, then we can solve consensus among all processes. That is, there exists a state $q \in S_n \subset Q$ such that $\mathcal{CN}(T_q) = n$. This is weaker than saying that for every state, we can solve consensus among n processes. In particular, it is not possible to reach such a state $q \in S_n$ in a wait-free manner, as we prove later.

Let us first define the sets S_k of synchronization states formally, then we will provide relevant bounds for the consensus number of an ERC20 token object in a synchronization state.

Enabled spenders. For every state $q = (\beta, \alpha) \in Q$, let $\sigma_q \colon \mathcal{A} \to 2^{\Pi}$ denote the mapping associating each account a to its *enabled spenders* according to q, i.e., the set of processes that are enabled to transfer tokens from account a w.r.t. balances β and allowances α specified by state q. Formally,

$$\sigma_q(a) = \{p \in \Pi : p = \omega(a) \ \vee \ \alpha(a, p) > 0\}. \tag{10}$$

Note that we explicitly include the account owner $\omega(a)$ in the set of enabled spenders for account a. We conventionally assume that an account with zero balance has only its owner as enabled spender, i.e., $\beta(a) = 0 \implies \sigma_q(a) = \{\omega(a)\}$. Indeed, even if there may be some process p, other than the owner, with positive allowance for account a, i.e., $\beta(a) = 0$ and $\alpha(a, p) > 0$, this process would not be able to transfer tokens from a unless the balance is increased.

State partition. Let Q_k, with $k \leq n$, be the set of states with exactly k valid spenders from the same account, i.e.,

$$Q_k = \{q \in Q : \max_{a \in \mathcal{A}} |\sigma_q(a)| = k\}. \tag{11}$$

Observe that the subsets $Q_1, \ldots, Q_n$ define a partition. Intuitively, we would like to say that each subset is associated to a given level of synchronization, defining a hierarchy $Q_1 \prec \cdots \prec Q_n$ reflecting the synchronization level, where Q_k corresponds to consensus number k. Importantly, the level of synchronization may change as the object's state is updated. In fact, for every k and for all states $q \in Q_k$, there exists a valid transition $(q, p, o, r, q') \in \Delta$, such that

$$q \in Q_k, \quad p = \omega(a), \quad o = \textit{approve}, \quad r = \text{TRUE}, \quad q' \in Q_{k+1}, \tag{12}$$

in the sense that it is possible to reach some state in Q_{k+1} from $q \in Q_k$. However, *the only way to do so* is by letting *the owner* of a k-spender account a approve a new spender.

Synchronization states. Later in this section, we show how to implement consensus from an ERC20 token object. Intuitively, we leverage an account for which multiple spenders have been approved: we let the spenders engage in a "race" where they compete for spending the account's tokens, and the "winner" of this competition gets to choose the decided value (in the consensus protocol). This idea crucially relies on the fact that there is a *unique* winner. To guarantee this, we need to impose an additional requirement on the balance and allowances of the account used in the implementation. We formally specify such requirement by defining predicate $\mathtt{U} \colon \mathcal{A} \times Q \to \{\text{TRUE}, \text{FALSE}\}$ (ensuring unique transfers) as follows. Namely, given a state $q = (\beta, \alpha)$ and an account a, we define:

$$\begin{aligned} \mathtt{U}(a, q) \quad \text{if and only if} \quad & \beta(a) > 0 \quad \wedge \\ & (|\sigma_q(a)| \leq 2 \quad \vee \quad \forall p_i, p_j \in \sigma_q(a) \setminus \{\omega(a)\} \ : \ \alpha(a, p_i) + \alpha(a, p_j) > \beta(a)) \,. \end{aligned} \tag{13}$$

We introduce further notation to identify relevant states which will appear in our main results. For every k as above, we define $S_k \subset Q_k$ to be the set of states q with exactly k valid spenders from the same account a and such that predicate $\mathtt{U}$ holds for (a, q):

$$S_k = \{q \in Q \ : \ \exists a \in \mathcal{A} \ : \ |\sigma_q(a)| = k \ \wedge \ \mathtt{U}(a, q)\} \tag{14}$$

We refer to the states in S_k as *k-synchronization states*. Intuitively, $q \in S_k$ are the states from which we can solve consensus for k processes, i.e., using an object type T_q, but not for more than k processes.

Theorem 2. *For every $k \leq n$ it holds $\mathcal{CN}(T_{S_k}) \geq k$.*

Proof. We show an implementation of a consensus object C for k processes, using an instance of a T_q object, with $q \in S_k$, and k atomic registers $R[1], \ldots, R[k]$. By the hypothesis $q \in S_k$, at least one account has k enabled spenders (cf. (14)) and satisfies the requirements defined by predicate `U` (defined in (13)) with respect to state $q = (\beta, \alpha)$. Without loss of generality, let $a_1 \in \mathcal{A}$ denote one such account, and let $\sigma_q(a_1) = \{p_1, \ldots, p_k\}$ with $p_1 = \omega(a_1)$. Let $B = \beta(a_1)$ and $A_j = \alpha(a_1, p_j)$, $j \in \{2, \ldots, k\}$, denote the balance of a_1, resp., the allowances of processes $p_2, \ldots, p_k$ for account a_1, w.r.t. state $q = (\beta, \alpha)$. Finally, let a_d be any account in $\{a_2, \ldots, a_k\}$. The code for the implementation is shown in Algorithm 1, and described below.

Algorithm 1 Wait-free implementation of a consensus object C among k processes in $\{p_1, \ldots, p_k\}$ using an ERC20 object T_q, with $q \in S_k$, associated to an account set $\mathcal{A} = \{a_1, \ldots, a_n\}$.

```
1:  State
2:      R[j] ← ⊥, j ∈ {1, ..., k}
3:      An ERC20 object T initialized such that:
4:          T.balances[a_1] = B
5:          T.allowances[a_1][p_j] = A_j, j ∈ {2, ..., k}
6:  operation propose(v) // Code for process p_i
7:      R[i].write(v)
8:      if p_i = p_1 then
9:          T.Transfer(a_d, B) // Transfer full balance
10:     else T.transferFrom(a_1, a_d, A_i)
11:     for j ∈ {2, ..., k} do
12:         if T.allowances(a_1, p_j) = 0 then
13:             return R[j].read()
14:     return R[1].read()
```

Briefly, each process p_i writes its proposed value v in a register $R[i]$. Then process p_1 attempts to transfer its whole balance B to account a_d, and process $p_i \neq p_1$ invokes operation T.*transferFrom* as an attempt to transfer its whole allowance A_i from a_1 to a_d. Since only one of the *transfer* and *transferFrom* invocations succeeds (as we prove shortly), we can safely decide the value proposed by the process which triggered the successful transfer. The intuition is that only the invocation of *transfer* by p_1 or the first completing invocation of *transferFrom* by some process p_{i^*}, for $i^* \in \{2, \ldots, k\}$, succeeds. Upon completion of that operation, no other process will be able to issue its own transfer because the balance of a_1 will be too low (this is guaranteed by the predicate `U` defined in (13)). Moreover, while the allowance of process p_{i^*} will be 0, the rest of the processes will still have positive allowances. Since the allowances can be read by all processes, every process can determine who won the competition and decide the value proposed by the winner. Therefore, once an operation *propose* completes by returning decision value v^*, every other process that invokes *propose* also decides v^*. More precisely, we select the "winner" process p_{i^*} as the one which succeeds in spending its allowance by transferring A_{i^*} tokens from a_1 to a_d. If none of the processes is found to have zero allowance, then p_1 must have been the first that called *propose*, and thus consumed the whole balance and caused any other calls to *propose* to fail.

We now show that the proposed implementation satisfies the *termination*, *validity*, and *agreement* properties of a consensus object (cf. Section 3). Regarding the termination property, observe that all instructions of operation C.*propose* do terminate: writing the proposed value to $R[i]$ terminates because of the use of an atomic register; the call to *transferFrom* terminates because it only involves reading from and writing to registers; the *for* loop is bounded by the number of processes k, and each iteration involves reading the allowance of a process p_j and potentially reading from the corresponding register $R[j]$ (termination follows by the properties of register R). The validity property holds because the decided value

is read from one of the registers $R[i]$ written by process p_i, for $i \in \{1, \ldots, k\}$, and the proposal of each process p_i is written before the *read* operation on that register is invoked (this is enforced by the *if* condition, cf. line 12). Hence, the decided value must be the proposal of some process p_j, for $j \in \{1, \ldots, k\}$. As for the consistency property, as we already mentioned, only the first invocation of operation *transfer* or *transferFrom* may succeed. In the former case, no invocation to *transferFrom* can ever succeed, hence no allowance can be set to 0, hence all processes will return the value proposed by p_1. In the latter case, the allowance of one of the processes p_{i^*}, for $i^* \in \{2, \ldots, k\}$ will be decreased from A_{i^*} to 0, and the *if* condition (cf. line 12) ensures that only the register written by a process with an allowance of 0 may be read. □

The previous theorem provides a lower bound for the consensus number of a token object T_q with initial state $q \in S_k$. Therefore, so far we can deduce the following inequalities (where the right-most inequality trivially holds):

$$k \overset{(Thm.2)}{\leq} \mathcal{CN}(T_{S_k}) \leq \infty \tag{15}$$

The upper bound in (15) is a loose one. We proceed with establishing a tight upper bound for the consensus number of T. Similarly to the case of the lower bound, we will need to condition our statement on the object's state.

Observe that starting from the initial state q_0 as defined in the original ERC20 specification—i.e., no process is authorized to issue transfers from accounts they do not own, and all but the contract deployer have zero balances (cf. Algorithm 3, Appendix A)—it is possible to reach a state $q \in S_k$ *as long as* tokens are transferred across accounts, and the owner of an account a with positive balance approves other $k-1$ spenders with sufficient allowances. Therefore, reaching a state in S_k is conditioned on all these $k-1$ *approve* operations succeeding, and ultimately on the account owner p_a not failing until then. Due to the above condition, a *wait-free* implementation of consensus from T_{q_0} is unachievable. More generally, starting from any state $q \in Q_k$, it is not possible to wait-free implement consensus among $k' > k$ processes, as we prove in the following theorem.

Theorem 3. *For every $k \leq n$ it holds $\mathcal{CN}(T_{Q_k}) \leq k$.*

Proof. We proceed by contradiction and assume a wait-free implementation of consensus for k' processes using objects of type T_{Q_k} and atomic registers, where $k' > k$, hence we show that for any such implementation there exists an infinite sequential execution that leaves it in a bivalent state.

Let us first recall some relevant terminology. A protocol state is *bivalent* if, starting from that state, there exists some execution in which the processes decide 0 and some execution in which they decide 1. A protocol state is called *critical* if it is bivalent and any subsequent state, reached by having a process invoke any of the object's methods, is univalent. Every wait-free consensus protocol has a critical state [18]. In the following, we denote one such state by q_c. Further, the invocation which brings the protocol from a critical state to a univalent state is called a *decision step*.

Without loss of generality, let $p_1, p_2 \in \Pi$ be processes such that the decision step for p_1, denoted by o_1, brings the protocol into a 0-valent state, and the decision step for p_2, denoted by o_2, brings it into a 1-valent state. The rest of the proof is case analysis of the methods which p_1 and p_2 execute in these decision steps.

Let us first assume that the decision step for p_1 is to invoke any operation on an atomic register, while the decision step for p_2 is to invoke any operation on a T_{Q_k} object. Starting from q_c, the sequential execution of o_1 followed by o_2 brings the protocol into a 0-valent state q_1, since p_1 took a step first. Instead, the sequential execution of o_2 followed by o_1 brings the protocol into a 1-valent state q_2, since p_2 took a step first. However, the states q_1 and q_2 are identical, because the two operations o_1 and o_2 commute, a contradiction.

Let us now assume that at least one of the invocations, say o_1, is on a read-only method. Consider the sequential execution starting from q_c, where p_1 executes o_1, then p_2 executes o_2, resulting in state q_1, and then p_2 runs alone and terminates. In this execution, p_2 must decide 0, because p_1 took a step first. Consider now the execution starting from q_c, where p_2 executes o_2, resulting in state q_2, and then p_2 runs

alone and terminates. In this execution, p_2 decides 1. However, the states q_1 and q_2 differ only in the internal values of p_1, since the latter invoked a read-only method, hence they are indistinguishable for p_2. Yet, p_1 decides a different value starting from q_1, respectively, q_2, a contradiction.

According to the commutativity and read-only arguments just described, the decision steps of p_1 and p_2 must operate on the same object and invoke a method that modifies the state of that object [18]. In the following, we examine all possible combinations for the decision steps, and whenever they commute, or are read-only, we refer to the arguments above to imply a contradiction.

Observe that the methods *totalSupply*, *balanceOf*, and *allowance* of the ERC20 token object are read-only, hence we do not examine them further. Moreover, if both o_1 and o_2 are *approve* invocations, or if one of them is an *approve* invocation and the other is a *transfer* invocation, then o_1 and o_2 commute and a contradiction is reached as shown above. We proceed by analyzing the remaining, non-trivial cases.

Case 1: both o_1 and o_2 are invocations to the transfer method. Since *transfer* withdraws tokens from the account of the calling process, o_1 and o_2 commute except for the case when $o_1 = \textit{transfer}(a_2, x)$, that is, a transfer of x tokens to the account of p_2, and the balance of p_2 is not sufficient to execute the transfer o_2 before o_1, that is, o_2 returns FALSE if executed before o_1. Observe that in this case, o_2 is equivalent to a read-only operation, therefore a contradiction is reached as described earlier. (For instance, consider the following two executions: in the first one, p_2 executes o_1 and then runs alone, deciding 0; in the second one, operation o_2 is executed first, followed by o_1, hence p_1 runs alone and decides 1.)

Case 2: both o_1 and o_2 are invocations to the transferFrom method. These invocations commute, except for the case when they both use the same source account a_s and the balance of a_s is only sufficient for one of the two transfers, and both processes are enabled to spend from a_s (without the latter condition the invocation would be equivalent to a read-only operation). Let us focus on this case. Since our implementation solves consensus among k' processes, and at most k processes are enabled spenders for the same account, where $k' > k$, there must be (at least) a process p_w that is not an enabled spender for account a_s—and by definition, p_w cannot be process $p_s = \omega(a_s)$. Assume wlog that the decision step o_3 taken by p_w brings the protocol in a 1-valent state (otherwise swap p_1 for p_2 in the following argument). Under this configuration, we will reach a contradiction for any possible method involved in o_3.

Let us begin with the case where o_3 is a *transferFrom* invocation with a_s as source account, as shown in Figure 1a. As process p_w is not enabled for account a_s, operation o_3 returns FALSE without modifying the state, thus it is equivalent to a read-only operation. Let us now consider the following two executions: process p_1 executes o_1, reaching state q_1, and then runs alone, thus deciding 0; process p_w executes o_3, then process p_1 executes o_1 reaching state q_3, then process p_1 runs alone, thus deciding 1. We have a contradiction, because states q_1 and q_3 are indistinguishable to process p_1.

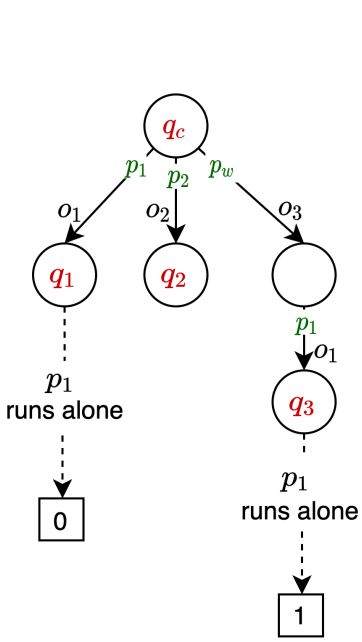
Moreover, if operation o_3 is a *transferFrom* invocation with source account a_t, with $t \neq s$, then operations o_1 and o_3 commute, and a contradiction is reached with a similar argument as above. A similar argument can be applied to all other possible methods, by observing that o_3 is either read-only (*totalSupply*, *balanceOf*, *allowance*), or it commutes with o_1 (*approve*, *transfer*), because $p_w \neq p_s$.

Case 3: operations o_1 and o_2 are a transfer, respectively, a transferFrom invocation, or vice versa. This case is analogous to the previous one. Indeed, if the *transferFrom* invocation has a source account other than a_1, then the two invocations commute, while if it has a_1 as source account, the same reasoning as in the previous case, making use of process p_w, applies.

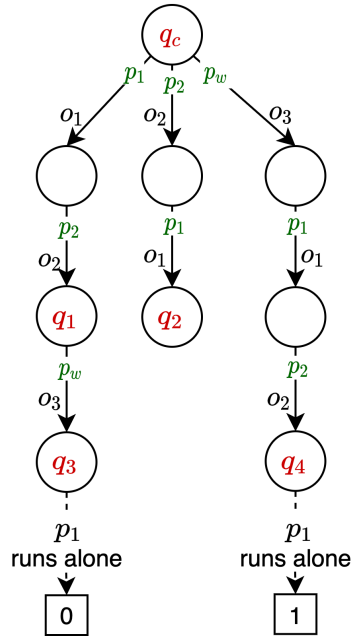
Case 4: operation o_1 is an approve invocation and o_2 is a transferFrom invocation. Let us examine the case where o_1 approves process p_2 and o_2 uses a_1 as source account—in all other cases, the two invocations commute. We distinguish two cases.

In the first case, assume that p_2 is not already an enabled spender for account a_1. Then operation o_2, if executed before o_1, returns FALSE and hence it does not affect the state of T_{Q_k}. Therefore, o_2 is equivalent to a read-only operation and a contradiction is reached with the exact same executions as in Case 1 (see above).

In the second case, assume that p_2 is already an enabled spender for account a_1. Then, as depicted in Figure 1b, the states q_1 and q_2, reached by the sequential execution of o_1 and then o_2, respectively, by the sequential execution of o_2 and then o_1, are not identical (hence we cannot deduce an immediate



(a) Case 2: both operations o_1 and o_2 are invocations to the *transferFrom* method.



(b) Case 4: operation o_1 is an *approve* invocation and o_2 is a *transferFrom* invocation.

Figure 1. Possible state transitions from the critical state q_c.

contradiction). However, in such case there must be a process p_w that is not an enabled spender for a_1 and, thus, every possible method invocation o_3 is either read-only or commutes with o_1 and o_2. Suppose the decision step taken by p_w brings the protocol in a 1-valent state (the reasoning for a 0-valent case is analogous). Then the sequential execution of operations o_1, o_2, and then o_3 results in a state q_3 from which p_1 decides 0. In contrast, the sequential execution of o_3 followed by o_1 and then o_2 results in a state q_4 from which p_1 decides 1. By observing that $q_3 = q_4$, we reach a contradiction. □

Putting it all together, we have:

$$k \overset{(Thm.2)}{\leq} \mathcal{CN}(T_{S_k}) \quad \text{and} \quad \mathcal{CN}(T_{Q_k}) \overset{(Thm.3)}{\leq} k. \tag{16}$$

Observing that $S_k \subseteq Q_k \implies \mathcal{CN}(T_{S_k}) \leq \mathcal{CN}(T_{Q_k})$, we can deduce exact synchronization requirements for T_q when q is a synchronization state, i.e., $q \in S_k$:

$$k \leq \mathcal{CN}(T_{S_k}) \leq \mathcal{CN}(T_{Q_k}) \leq k \quad \implies \quad \mathcal{CN}(T_{S_k}) = k. \tag{17}$$

Notice that successful completion of specific *approve* operations is *necessary* to reach a synchronization state q from which we can wait-free implement consensus for arbitrarily many processes. Concretely, reaching a state $q' \in Q_{k'}$ starting from any state $q \in Q_k$, with $k' > k$, requires the owner of some account with k enabled spenders to approve other $k' - k - 1$ spenders. If such *approve* operations—which change the number of enabled spenders for the same account—were not enabled, then the resulting token object would be no stronger than the k-shared asset transfer object. To make this argument formal, we define an auxiliary token object $T|_{Q_k}$ by restricting the ERC20 token T_q, with $q \in Q_k$, so that only transitions within Q_k are permitted. In other words, $T|_{Q_k}$ is a restricted version of T which does not allow any transition to a state $q' \in Q \setminus Q_k$. The resulting token object reduces to k-shared asset transfer, as we show next.

Theorem 4. *For every $k \leq n$, there exists a wait-free implementation of a token object $T|_{Q_k}$ from objects of type k-AT and atomic registers. In particular, $\mathcal{CN}(T|_{Q_k}) \leq \mathcal{CN}(k\text{-}AT) = k$.*

Proof. We present a wait-free implementation of object $T|_{Q_k}$ from a k-shared asset transfer object k-AT (cf. Definition 1), which is known to have consensus number k [16], and atomic registers. The result is implied by the inequality $\mathcal{CN}(T|_{Q_k}) \leq \mathcal{CN}(k\text{-}AT)$, which follows from Theorem 1.

Intuitively, for $q \in Q_k$, we envision every account a that has k enabled spenders w.r.t. q as a *k-shared* account, so that we can emulate the methods of object T by invoking the methods of k-AT and using atomic registers. Indeed, the token object T is similar to k-shared asset transfer, the crucial difference being that each account owner can approve new spenders for their account and, therefore, can in principle realize k'-shared accounts with $k' > k$. However, the latter operations are disabled for the "restricted" object $T|_{Q_k}$, hence all operations can be simulated by the methods of k-AT and with registers.

We provide an explicit implementation in Algorithm 2. The balances k-$AT.\beta$ and the owner map k-$AT.\mu$ are initialized in lines 3 and 4, respectively, according to the state q. For each account $a \in \mathcal{A}$, the algorithm makes use of n atomic registers $R_a[j]$, for $1 \leq j \leq n$, to keep track of the allowance that the owner of a has assigned to process p_j. These registers are initialized in line 6 according to the allowance of each account in state q.

Recall that the owner map k-$AT.\mu$ of an asset transfer object is static, i.e., it is defined upon creation of the object. Hence, in order to keep track of dynamically evolving allowances for $T|_{Q_k}$, we make use of multiple instances of a k-AT object: whenever the set of enabled spenders for a given account changes (and as long as the account has no more than k enabled spenders), we create a new instance of the k-AT object, with the same balances as the previous instance and an owner map reflecting the updated allowances. In the pseudocode, this is expressed by updating k-$AT.\mu$ in lines 21–23.

To see why the implementation is correct, observe that no process can transfer more tokens than it has; this is ensured by making use of the k-AT object. Moreover, for every account a we keep track of the allowances using registers $R_a[j]$, for $j \in \{1, \ldots, n\}$, and the corresponding balances are managed by the k-AT object.

Notice that by definition of $T|_{Q_k}$, all *approve* operations are restricted to transitions within Q_k. Therefore, all valid configurations of account balances and allowances will enable at most k spenders for the same account. This allows treating such accounts as k-shared, and ultimately enables a correct simulation with the methods of the asset transfer object. The implementation is wait-free because we do not make use of any *for* loop, every method is constructed without the need to wait for other processes to complete their operations and the k-AT object is wait-free. □

ERC20 token *vs* k-shared asset transfer. Informally, Theorem 2 and Theorem 4 jointly confirm the intuition that the ERC20 token object is more complex than, yet uncomparable with, the k-shared asset transfer object. On the one hand, the ERC20 token object is similar to k-shared asset transfer, in the sense that k-shared accounts can be emulated, to a certain extent, having an account owner approving sufficiently many spenders. On the other hand, for ERC20 tokens any synchronization level can be reached, in principle, by enabling sufficiently many spenders for the same account, suggesting that ERC20 tokens are strictly more powerful, in terms of synchronization level, than k-AT. Indeed, while the owners of a shared account must be fixed upfront when the contract is deployed, the enabled spenders for an ERC20 account can change dynamically, as the account owner wishes. Similarly, the amount of tokens that each enabled spender is allowed to transfer from that account is flexibly chosen, and can be modified at any time, by the account owner. This is in sharp contrast with k-shared asset transfer objects, as the latter has a static consensus number. Nevertheless, increasing the synchronization power in ERC20 tokens cannot be done in a wait-free manner.

6 Extension to Other Token Standards

In this section, we discuss how to extend our results to other token standards on Ethereum beyond ERC20. As of the time of writing, several token implementations have been proposed within the Ethereum project, ERC20 being the major reference among all. Some of these proposals are in a testing phase while others

Algorithm 2 Wait-free implementation of a token object $T|_{Q_k}$, with initial state $q = (\beta, \alpha) \in Q_k$, from k-shared asset transfer objects $k\text{-}AT$. Code for process p_i.

State
  for $a \in \mathcal{A}$ **do**
    $k\text{-}AT.\beta[a] \leftarrow \beta(a)$ // Balance of account a in state q
    $k\text{-}AT.\mu[a] \leftarrow \sigma_q(a)$ // The enabled spenders of account a in state q
    for $p_j \in \Pi$ **do**
      $R_a[j] \leftarrow \alpha(a, p_j)$ // Allowance of account a to process p_j in state q

operation *transferFrom*$(a_s, a_d, value)$ // Transfer *value* from source a_s to destination a_d
  if $R_{a_s}[i] < value$ **then**
    return FALSE
  $R_{a_s}[i] \mathrel{-}= value$
  $k\text{-}AT.$*transfer*$(a_s, a_d, value)$

operation *transfer*$(a_d, value)$ // Transfer *value* from source a_i to destination a_d
  return $k\text{-}AT.$*transfer*$(a_{p_i}, a_d, value)$

operation *balanceOf*(a) // Read balance of a
  return $k\text{-}AT.$*balanceOf*(a)

operation *approve*$(p_j, value)$ // Approve spender p_j for account a_i
  if $|\{p_a\} \cup \{p_j \in \Pi : R_a[j] > 0\}| = k$ **then**
    return FALSE // Ensure we stay in Q_k
  $oldValue \leftarrow R_{a_i}[j]$
  $R_{a_i}[j] \leftarrow value$
  if $oldValue = 0$ **and** $value > 0$ **then**
    for $a \in \mathcal{A}$ **do**
      $k\text{-}AT.\mu[a] \leftarrow \{p_a\} \cup \{p_j \in \Pi : R_a[j] > 0\}$
  return TRUE

operation *allowance*(a, p_j) // Read allowance of p_j for account a
  return $R_a[j]$

operation *totalSupply*()
  return $\sum_{a \in \mathcal{A}} k\text{-}AT.\beta[a]$

have already reached a final phase and have been adopted [1]. We overview the proposals that have reached the final stage.

The ERC777 token standard aims to solve some problems related to ERC20, while maintaining backward compatibility [8]. It defines new features, some of which are similar to those of ERC20, to interact with the tokens. In particular, it defines *operators* to transfer tokens on behalf of another address, similarly to the mechanism enabled by the *allowances* in ERC20, and *hooks*, to simplify the sending process and to offer a single way for sending tokens to any recipient. One of the main differences compared to ERC20 is the mechanism of allowing processes to manage tokens on behalf of others. In ERC20, the *approve* method lets an account owner p define an amount of tokens that the approved process p' is allowed to spend on behalf of p. In contrast, an *operator* p' in ERC777 is allowed to spend all the tokens owned by the approving process p. Nevertheless, it is immediate to extend our results to ERC777. Specifically, both Algorithms 1 and 2 can be adapted by replacing the approved spenders with the corresponding operators.

The ERC721 standard is inspired by ERC20, however, it provides an interface for *non-fungible* tokens [10]. In contrast to standard tokens, all non-fungible tokens are *unique*. In ERC721, every token is uniquely determined by an identifier *tokenId* and can be individually transferred using a *transferFrom* method. Similarly to ERC20, an account owner p can approve other processes to spend tokens on its behalf by invoking the *approve* method, specifying the process p' to be approved and a token identifier *tokenId*. We do not discuss the other methods specified by ERC721, as they fall outside the scope of this work. Although ERC721 defines tokens of different nature compared to ERC20 tokens, we notice that our techniques and results can also be applied, with some adjustment, to this standard. Concretely, Algorithm 1 can be adapted so that it uses a *specific token*, determined by its identifier *tokenId*, which all the participating processes are approved to spend; the winner of this race can then be determined by invoking *ownerOf* with token identifier *tokenId*. Modifying Algorithm 2 to match the ERC721 specification requires more care. More generally, implementing an ERC721 token object from k-AT appears challenging, if not impossible, as each ERC721 token is transferred individually rather than collectively, as is the case with fungible tokens. Instead of relying on k-AT, however, a series of k-consensus instances could be used, with each instance associated to an ERC721 token, so that k-consensus can be invoked each time a token is spent.

ERC1155 defines a smart-contract interface for managing multiple token types. In particular, it specifies methods that enable the execution of a number of transactions, possibly on different token types, or involving various source and target accounts, within a single method-call. While it is plausible that ERC1155 tokens inherit the synchronization requirements of ERC20 tokens, establishing formal requirements would need an in-depth analysis, based on combinations of accounts, which goes beyond the scope of this work.

Finally, the *Payable Token* standard ERC1363 follows the *approve* and *transferFrom* paradigm of ERC20 tokens, but adds a layer of indirection. Specifically, it allows processes to specify *arbitrary code*, which is executed upon receiving a token through *transfer*, *transferFrom*, or upon completion of an *approve* operation. The possibility of executing an arbitrary contracts precludes establishing exact synchronization requirements *a priori*, as this can be arbitrary.

7 Conclusion and future directions

Prior work shows that the asset transfer object, providing the basic functionality of a cryptocurrency, has consensus number 1 [16]. This means that implementing a "plain" cryptocurrency such as Bitcoin [21] does not require synchronization among processes, and hence the consensus layer of Bitcoin could be replaced by a fully asynchronous dissemination protocol, which does not order transactions. This important result however does not apply to blockchains with richer smart-contract support such as Ethereum. In fact, enabling the execution of *arbitrary* smart contracts requires agreement among all blockchain nodes. Nevertheless, it remains open whether *specific* smart contracts need consensus or not, and more generally, which level of synchronization they require.

In this work, we analyze the synchronization requirements of one such smart contract—the ERC20 token standard of Ethereum—through the lens of wait-free implementations, establishing the consensus number of an associated shared-memory token object. Our results show that an ERC20 token contract may require different levels of synchronization, depending on its state configurations. In other words, the ERC20 token object has a *dynamic* consensus number: when initialized according to the standard [28], its consensus number is 1; however, as soon as an account owner approves other spenders for its account, the consensus number of the object may increase. In fact, there exist executions that modify the state so that the consensus number becomes k, for every k with $1 \leq k \leq n$.

Our results imply that while executing arbitrary smart contracts requires consensus among *all* processes, synchronizing a dedicated subset of participants is sufficient for realistic applications such as token contracts. In the case of ERC20 tokens, consensus indeed only needs to be reached among the largest set $\sigma_q(a)$ of enabled spenders for the same account a; importantly, the exact synchronization requirements can be readily deduced from the current object's state q by reading the current balances and allowances. This insight opens up the possibility to deploy realistic smart contracts, such as ERC20 tokens, on more scalable and performant protocols than consensus-based blockchains. Namely, the consistency mechanism could be flexibly adapted, during execution, to require higher or lower coordination among nodes depending on the current state of the smart contract, so that only the minimal synchronization requirements are matched.

We suggest as an interesting open problem to develop distributed protocols meeting the dynamic synchronization requirements of ERC20 tokens. Such protocols could replace the consensus layer of traditional blockchain platforms with a more efficient broadcast method, as shown earlier for asset transfer [6]. This would generally work under asynchrony and yet provide an atomic broadcast functionality among every account owner and its enabled spenders.

References

[1] "Ethereum Request for Comments." https://eips.ethereum.org/erc.

[2] A. Back, M. Corallo, L. Dashjr, M. Friedenbach, G. Maxwell, A. Miller, A. Poelstra, J. Timón, and P. Wuille, "Enabling blockchain innovations with pegged sidechains." https://blockstream.com/sidechains.pdf.

[3] E. Buchman, J. Kwon, and Z. Milosevic, "The latest gossip on BFT consensus," *CoRR*, vol. abs/1807.04938, 2018.

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

[5] C. Cachin, B. Junker, and A. Sorniotti, "On limitations of using cloud storage for data replication," in *DSN Workshops*, pp. 1–6, IEEE Computer Society, 2012.

[6] D. Collins, R. Guerraoui, J. Komatovic, P. Kuznetsov, M. Monti, M. Pavlovic, Y. A. Pignolet, D. Seredinschi, A. Tonkikh, and A. Xygkis, "Online payments by merely broadcasting messages," in *DSN*, pp. 26–38, IEEE, 2020.

[7] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. E. Kosba, A. Miller, P. Saxena, E. Shi, E. G. Sirer, D. Song, and R. Wattenhofer, "On scaling decentralized blockchains - (A position paper)," in *Financial Cryptography Workshops*, vol. 9604 of *Lecture Notes in Computer Science*, pp. 106–125, Springer, 2016.

[8] J. Dafflon, J. Baylina, and T. Shababi, "EIP-777: ERC777 Token Standard." https://eips.ethereum.org/EIPS/eip-777.

[9] S. Dziembowski, L. Eckey, S. Faust, and D. Malinowski, "Perun: Virtual payment hubs over cryptocurrencies," in *IEEE Symposium on Security and Privacy*, pp. 106–123, IEEE, 2019.

[10] W. Entriken, D. Shirley, J. Evans, and N. Sachs, "EIP-721: ERC-721 Non-Fungible Token Standard." https://eips.ethereum.org/EIPS/eip-721, 2018.

[11] Ethereum Foundation, "Ethereum." https://ethereum.org/.

[12] Ethereum Foundation, "Ethereum 2.0." https://ethereum.org/en/eth2/.

[13] M. J. Fischer, N. A. Lynch, and M. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[14] J. A. Garay, A. Kiayias, and N. Leonardos, "The bitcoin backbone protocol: Analysis and applications," in *EUROCRYPT (2)*, vol. 9057 of *Lecture Notes in Computer Science*, pp. 281–310, Springer, 2015.

[15] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *SOSP*, pp. 51–68, ACM, 2017.

[16] R. Guerraoui, P. Kuznetsov, M. Monti, M. Pavlovic, and D. Seredinschi, "The consensus number of a cryptocurrency," in *PODC*, pp. 307–316, ACM, 2019.

[17] R. Guerraoui, P. Kuznetsov, M. Monti, M. Pavlovic, and D. Seredinschi, "Scalable byzantine reliable broadcast," in *DISC*, vol. 146 of *LIPIcs*, pp. 22:1–22:16, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[18] M. Herlihy, "Wait-free synchronization," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 1, pp. 124–149, 1991.

[19] IOTA Foundation, "Iota." https://www.iota.org/.

[20] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, "Omniledger: A secure, scale-out, decentralized ledger via sharding," in *IEEE Symposium on Security and Privacy*, pp. 583–598, IEEE Computer Society, 2018.

[21] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System." Whitepaper, http://bitcoin.org/bitcoin.pdf, 2009.

[22] J. Poon and T. Dryja, "The bitcoin lightning network: Scalable off-chain instant payments." https://lightning.network/lightning-network-paper.pdf.

[23] M. Raynal, *Concurrent Programming - Algorithms, Principles, and Foundations*. Springer, 2013.

[24] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, 1990.

[25] Y. Sompolinsky and A. Zohar, "PHANTOM: A scalable blockdag protocol," *IACR Cryptol. ePrint Arch.*, vol. 2018, p. 104, 2018.

[26] T. Swanson, "Consensus-as-a-service: A brief report on the emergence of permissioned, distributed ledger systems." Available online, http://www.ofnumbers.com/wp-content/uploads/2015/04/Permissioned-distributed-ledgers.pdf, 2015.

[27] F. Victor and B. K. Lüders, "Measuring ethereum-based ERC20 token networks," in *Financial Cryptography*, vol. 11598 of *Lecture Notes in Computer Science*, pp. 113–129, Springer, 2019.

[28] F. Vogelsteller and V. Buterin, "EIP-20: ERC-20 Token Standard." Available online, https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md, 2015.

[29] Web3 Foundation, "Polkadot." https://polkadot.network/.

[30] M. Yin, D. Malkhi, M. K. Reiter, G. Golan-Gueta, and I. Abraham, "Hotstuff: BFT consensus with linearity and responsiveness," in *PODC*, pp. 347–356, ACM, 2019.

[31] M. Zamani, M. Movahedi, and M. Raykova, "Rapidchain: Scaling blockchain via full sharding," in *ACM Conference on Computer and Communications Security*, pp. 931–948, ACM, 2018.

[32] Q. Zhou, H. Huang, Z. Zheng, and J. Bian, "Solutions to scalability of blockchain: A survey," *IEEE Access*, vol. 8, pp. 16440–16455, 2020.

A ERC20 Token Standard

Algorithm 3 Sequential specification of ERC20 functionalities.

state
    const d, the process that deployed the contract, used only in the initialization of *balances*
    const string name
    const string symbol
    const int decimals
    const int totalSupply
    balances [] $\subseteq \mathcal{A} \times \mathbb{N}$, initially $balances[d] = totalSupply, balances[i] = 0$ for $i \neq d$
    allowances [][] $\subseteq \mathcal{A} \times \mathcal{P} \times \mathbb{N}$, initially $\emptyset$

operation *totalSupply*()
    return *totalSupply*

operation *balanceOf*(*owner*)
    return *balances*[*owner*]

operation *transfer*(*to*, *value*) // Code for process p_i
    if $balances[p_i] < value$ **then**
        return FALSE
    else
        $balances[p_i] -= value$
        $balances[to] += value$
        return TRUE

operation *transferFrom*(*from*, *to*, *value*) // Code for process p_i
    if $allowances[from][p_i] < value$ **then**
        return FALSE
    else if $balances[from] < value$ **then**
        return FALSE
    else
        $allowances[from][p_i] -= value$
        $balances[from] -= value$
        $balances[to] += value$
        return TRUE

operation *approve*(*spender*, *value*) // Code for process p_i
    $allowances[p_i][spender] \leftarrow value$
    return TRUE

operation *allowance*(*owner*, $spender$)
    return *allowances*[*owner*][*spender*]