A Tight Bound on Multiple Spending in Decentralized Cryptocurrencies

João Paulo Bezerra
Télécom Paris, Institut Polytechnique de Paris, France

Petr Kuznetsov
Télécom Paris, Institut Polytechnique de Paris, France

Abstract

The last decade has seen a variety of Asset-Transfer systems designed for decentralized environments. The major problem these systems address is *double-spending*, and solving it inherently imposes strong *trust* assumptions on the system participants. In this paper, we take a non-orthodox approach to the double-spending problem that might suit better realistic environments in which these systems are to be deployed. We consider the *decentralized trust* setting, where each user may independently choose who to trust by forming their local quorums. In this setting, we define *k-Spending Asset Transfer*, a relaxed version of asset transfer which bounds the number of times a system participant may spend an asset it received. We establish a precise relationship between the decentralized trust assumptions and k, the optimal *spending number* of the system.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases Quorum systems, decentralized trust, consistency measure, asset transfer, accountability

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2023.31

Related Version The full version of the paper is available as a technical report.
Full Version: https://arxiv.org/abs/2205.14076

Acknowledgements This work was supported by TrustShare Innovation Chair.

1 Introduction

Fault models and quorum systems. Distributed protocols, such as consensus and broadcast, are generally built to be resilient against arbitrary (Byzantine) faults of system members. To maintain consistency and progress, these protocols typically have to assume that only a certain fraction of system members are allowed to be Byzantine. In the special case of a *uniform* fault model, where faults of system members are identically and independently distributed, bounds on the number f of Byzantine members that can be tolerated are well known: less than half of system members ($f < n/2$) in synchronous networks (using digital signatures) [27], and less than one third ($f < n/3$) in asynchronous or partially synchronous networks [7].

More general fault models can be captured via *quorum systems* [33, 40], collections of subsets of system participants, called *quorums*, that meet two conditions: in every system run, *every* two quorums should have at least one correct participant in common and *some* quorum should only contain correct participants. Intuitively, quorums encapsulate *trust* the system members express to each other. Every quorum can act on behalf of the whole system: an update or a query on the data is considered safe if it involves a trusted set of replicas.

Decentralized quorums. Conventionally, quorum assumptions are centralized: all participants share the same quorum system. In some large-scale distributed systems, it might be, however, difficult to expect that all participants come to the same trust assumptions.

27th International Conference on Principles of Distributed Systems (OPODIS 2023).
Editors: Alysson Bessani, Xavier Défago, Junya Nakamura, Koichi Wada, and Yukiko Yamauchi; Article No. 31; pp. 31:1–31:19
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Recently, the quorum-based approach to system design has been explored in a completely new way. It started with system implementations [37, 34] that allowed their users to not necessarily hold the same assumptions of who to trust, i.e., to maintain *local* quorum systems. Based on its local knowledge, a system member might have its own idea about which subsets of other participants are trustworthy and which are not. We come, therefore, to the model of *decentralized quorums*: each system member maintains its own quorum system.

Great effort has been invested into improving protocols designed for uniform fault models [12, 28, 21, 42], or in understanding which conditions on individual quorum systems are necessary and sufficient, so that some well-defined subset of participants can solve a problem [19, 8, 32]. However, little is understood about the "damage" that Byzantine processes might cause if these conditions do not hold, e.g., in the decentralized quorum system model. Intuitively, the more Byzantine processes there are or more strategically they are located in decentralized quorums, the more important is the impact they have on the system's consistency. But what exactly does "more important" mean here?

Asynchronous cryptocurrencies. In this paper, we study this question on the example of asset-transfer systems (or *cryptocurrencies*). Conventionally, the major challenge addressed by a cryptocurrency is to prevent *double spending*, when a malicious or misconfigured user manages to spend the same coin more than once. As was originally claimed by Nakamoto [36], preventing double spending in systems with mutual distrust requires honest users to agree on the order in which the transactions must be executed, i.e., to solve the fundamental problem of *consensus* [18]. Bitcoin achieves probabilistic *permissionless* consensus assuming a synchronous system and using the proof-of-work mechanism. The protocol is notoriously energy-consuming and slow. Since then, a long line of systems used consensus for implementing cryptocurrencies in both permissionless and permissioned contexts.

It has been later observed that cryptocurrencies do not always require consensus in general [23, 22]. It turns out that it is not always necessary to maintain a totally ordered set of transactions, a specific partial order may suffice. Intuitively, if we assume that each account has a single dedicated owner, it is sufficient to agree on the order of outgoing transactions *per account.* Transactions operating on different accounts can be ordered arbitrarily without affecting correctness. Double spending is excluded, as no user can publish "conflicting" transactions on its account (spending more money than its account holds). Recently proposed asynchronous (*consensus-free*) cryptocurrencies [4, 15] exhibit significant advantages over consensus-based protocols in terms of scalability, performance and robustness. However, as they still rely on classical quorum systems, they are challenging to apply at a large scale.

Contributions. In this paper, we explore the potential of *decentralized* quorums in implementing asynchronous cryptocurrencies. Naturally, this model allows us to formally capture the *double spending* phenomenon. In a way, we mimic the principle followed by real-world financial systems, where double spending is a routine phenomenon.

We introduce *k-Spending Asset Transfer*, a relaxed cryptocurrency abstraction suitable for decentralized trust models. Notice that in this model, quorums chosen by correct processes might not be *globally consistent*, i.e., some quorums might not overlap on a correct process. Byzantine processes can exploit this lack of consistency by enforcing correct processes to accept conflicting transactions with the same input, resulting in *multiple spending.*

Intuitively, a k-spending asset-transfer system guarantees that once a participant receives an asset, it spend it at most k times. It ensures, however, that any instance of multiple-spending that affects correct participants should be eventually detected and a proof of misbehaviour against the Byzantine process should be published.

As a bold analogy, one can think of a global financial trading system, where every national economy benefits from mutual trust, while cross-border interactions are less reliable. But if the lack of trust is exploited by a cheating trader, correct participants should eventually be able to detect and punish the cheater by, e.g., excluding it from the system.

We show how the parameter k in k-spending asset transfer relates to the structure of the underlying quorum assumptions. We visualize these assumptions via a family $\mathcal{G}_{\mathcal{Q},\mathcal{F}}$ of graphs, one for each possible faulty set $F \in \mathcal{F}$ and each quorum map S, mapping each process p to an element in its local quorum system $\mathcal{Q}(p)$. It turns out that the optimal number of times a coin can be spent in this system is precisely the maximum *independence number* over graphs in $\mathcal{G}_{\mathcal{Q},\mathcal{F}}$.

Thus, our contributions are three-fold. We introduce the abstraction of k-spending asset transfer that defines a precise bound k on the number of times a given asset can be spent, once it is received by a system participant. We represent decentralized trust assumption in the form of a family of trust graphs and show that its maximum independence number gives a lower bound on k. We present a k-asset transfer implementation that shows that the bound is tight. In addition, the algorithm maintains an accountability mechanism that keeps track of multiple spending and publishes evidences of misbehavior.

Road map. The rest of the paper is organized as follows. In Section 2 we present our system model. Section 3 introduces a graph representation of trust, used later in the paper to prove lower bounds on "the amount of inconsistency" in cryptocurrency implementations. In Section 4 we give the specification of *k-Spending Asset Transfer* (k-AT) and present a protocol for implementing it. We show that our k-AT algorithm is optimal in Section 5, by relating it to a relaxed broadcast abstraction: *k-Consistent Broadcast* (k-CB). We overview related work in Section 6. Finally, we discuss the results and future work in Section 7.

2 System Model

Processes. A system is composed of a set of *processes* $\Pi = \{p_1, ..., p_n\}$. Every process is assigned an *algorithm* (we also say *protocol*), an automaton defined as a set of possible *states* (including the *initial state*), a set of *events* it can produce and a transition function that maps each state to a corresponding new state. An event is either an *input* (a call operation from the application or a message received from another process) or an *output* (a response to an application call or a message sent to another process); *send* and *receive* denote events involving communication between processes.

Executions and failures. A *configuration* C is a collection of states of all processes. In addition, C^0 is used to denote a special configuration where processes are in their initial states. An *execution* (or a *run*) Σ is a sequence of events, where every event is associated with a distinct process and every *receive*(m) event has a preceding matching *send*(m) event. A process *misbehaves* in a run (we also call it *Byzantine*) if it produces an event that is not prescribed by the assigned protocol, given the preceding sequence of events, starting from the initial configuration C^0. If a process does not misbehave, we call it *benign*. In an infinite run, a process *crashes* if it prematurely stops producing events required by the protocol; if a process is benign and never crashes we call it *correct*, and it is *faulty* otherwise. Let *part*(Σ) denote the set of processes that produce events in an execution Σ.

Channels. Every pair of processes communicate over a *reliable channel*: in every infinite run, if a correct process p sends a message m to a correct process q, m eventually arrives, and q receives a message from p only if p sent it. We impose no synchrony assumptions. In particular, we assume no bounds on the time required to convey a message from one correct process to another.

Digital signatures. We use asymmetric cryptographic tools: a pair public-key/private-key is associated with every process in Π [9]. The private key remains secret to its owner and can be used to produce a *signature* for a statement, while the public key is known by all processes and is used to *verify* that a signature is valid. Every process have access to operations *sign* and *verify*: *sign* takes the process' identifier and a bit string as parameters and returns a signature, while *verify* takes the process' identifier, a bit string and a signature as parameters and return $b \in \{TRUE, FALSE\}$. We assume a computationally bound adversary: no process can forge the signature for a statement of a benign process.

Trust assumptions. We now define our decentralized trust model. A *quorum system map* $\mathcal{Q} : \Pi \to 2^{2^{\Pi}}$ provides every process with a set of process subsets: for every process p, $\mathcal{Q}(p)$ is the set of *quorums of* p. We assume that p includes itself in each of its quorums: $\forall Q \in \mathcal{Q}(p) : p \in Q$. Intuitively, $\mathcal{Q}(p)$ consists of sets of processes p expects to appear correct in system runs. From p's perspective, for every quorum $Q \in \mathcal{Q}(p)$, there must be an execution in which Q is precisely the set of correct processes. However, these expectations may be violated by the environment. We therefore introduce a *fault model* $\mathcal{F} \subseteq 2^{\Pi}$ (sometimes also called an *adversary structure*) stipulating which process subsets can be faulty. We assume *inclusion-closed* fault models that, intuitively, do not force processes to fail: $\forall F \in \mathcal{F},\ F' \subseteq F :\ F' \in \mathcal{F}$. From now on, we consider only executions Σ that *complies with* $\mathcal{F}$, i.e., the set of faulty processes in Σ is in $\mathcal{F}$.

Given a faulty set $F \in \mathcal{F}$, a process p is called *live under* F if it has a *live quorum*, i.e., $\exists Q \in \mathcal{Q}(p) : Q \cap F = \emptyset$. For example, let the uniform f*-resilient* fault model: $\mathcal{F} = \{F \subseteq \Pi : |F| \leq f\}$. If $\mathcal{Q}(p)$ includes all sets of $n - f$ processes, then p is guaranteed to have at least one live quorum in every execution. On the other hand, if $\mathcal{Q}(p)$ expects that a selected process $q \neq p$ is always correct ($q \in \cap_{Q \in \mathcal{Q}(p)} Q$), then p is not live in any execution with a faulty set such that $q \in F$.

In the rest of the paper, we consider a *trust model* $(\mathcal{Q}, \mathcal{F})$, where $\mathcal{Q}$ is a quorum map and $\mathcal{F}$ is a fault model.

3 Graph Representation of Trust

We use undirected graphs to depict possible scenarios of executions with trust assumptions $(\mathcal{Q}, \mathcal{F})$. Intuitively, each graph represents a situation where a correct process hears from a quorum before accepting a statement in a protocol. Let $S : \Pi \to 2^{\Pi}, S(p) \in \mathcal{Q}(p)$, be a map providing each process with one of its quorums, and $\mathcal{S}$ be the family of all possible such maps. For a fixed faulty set $F \in \mathcal{F}$ and $S \in \mathcal{S}$, the graph $G_{F,S}$ is a tuple $(\Pi_F, E_{F,S})$ where:

- $\Pi_F = \Pi - F$, i.e., the set of correct processes;
- Nodes p and q are connected with an edge *iff* their quorums $S(p)$ and $S(q)$ intersect in a correct process, i.e., $(p, q) \in E_{F,S} \Leftrightarrow S(p) \cap S(q) \not\subseteq F$.

▶ **Example 1.** Consider a system of four processes, where $\Pi = \{p_1, p_2, p_3, p_4\}$, $\mathcal{F} = \{\{p_3\}\}$, and the individual quorum systems are:

$$\mathcal{Q}(p1) = \{\{p_1, p_2, p_3\}\} \qquad \mathcal{Q}(p2) = \{\{p_1, p_2\}, \{p_2, p_4\}\}$$

$$\mathcal{Q}(p3) = \{\{p_1, p_2, p_4\}\} \qquad \mathcal{Q}(p4) = \{\{p_2, p_4\}, \{p_3, p_4\}\}$$

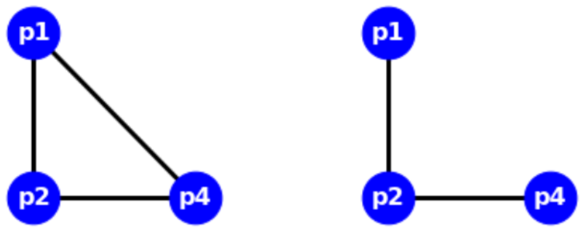

Figure 1 Graph structures of Example 1: G_{F,S_1} and G_{F,S_2} respectively.

Consider an execution with $F = \{p_3\}$, the set of correct processes Π_F is $\{p_1, p_2, p_4\}$. Let $S_1 \in \mathcal{S}$ be a quorum map for $\mathcal{Q}$ such that $S_1(p_1) = \{p_1, p_2, p_3\}$, $S_1(p_2) = \{p_1, p_2\}$ and $S_1(p_4) = \{p_2, p_4\}$. Every pair of nodes in the resulting graph G_{F,S_1} have quorums intersecting in $p_2 \in \Pi_F$, resulting in a fully connected graph. Now let $S_2 \in \mathcal{S}$ be a quorum map such that $S_2(p_1) = \{p_1, p_2, p_3\}$, $S_2(p_2) = \{p_2, p_4\}$ and $S_2(p_4) = \{p_3, p_4\}$. Since $S_2(p_1) \cap S_2(p_4) \subseteq F$, the resulting graph G_{F,S_2} has a missing edge. Figure 1 depicts G_{F,S_1} and G_{F,S_2}.

Inconsistency number. We recall two useful definitions from graph theory: *Independent Set* and *Independence Number*.

▶ **Definition 2** (Independent Set). *A set $C \subseteq V$ is an* independent set *of $G = (V, E)$ iff no pair of nodes in C is adjacent, i.e., $\forall p, q \in C : (p, q) \notin E$. C is* maximum *iff for every independent set C' of G: $|C'| \leq |C|$.*

▶ **Definition 3** (Independence Number). *The* independence number *of G is the size of its maximum independent set(s).*

Given the pair $(\mathcal{Q}, \mathcal{F})$, we note $\mathcal{G}_{\mathcal{Q},\mathcal{F}}$ the family of graphs including all possible $G_{F,S}$, where $F \in \mathcal{F}$ and $S \in \mathcal{S}$.

▶ **Definition 4** (Inconsistency Number). *The* inconsistency number *of $(\mathcal{Q}, \mathcal{F})$ is the highest independence number among all $G_{F,S} \in \mathcal{G}_{\mathcal{Q},\mathcal{F}}$. Formally, Let $\mu : \mathcal{G}_{\mathcal{Q},\mathcal{F}} \to N$ map each $G_{F,S} \in \mathcal{G}_{\mathcal{Q},\mathcal{F}}$ to its independence number $\lambda(\mathcal{G}_{\mathcal{Q},\mathcal{F}})$, then $\lambda(\mathcal{G}_{\mathcal{Q},\mathcal{F}}) = max(\{\mu(G_{F,S}) | G_{F,S} \in \mathcal{G}_{\mathcal{Q},\mathcal{F}}\})$.*

▶ **Example 5.** Coming back to Example 1, the graph G_{F,S_1} is fully connected, thus it has independence number 1. On the other hand, the maximum independent set in G_{F,S_2} is $\{p1, p4\}$, as a result, G_{F,S_2} has independence number 2. Now consider a pair $(\mathcal{Q}, \mathcal{F})$ where $\mathcal{Q}$ and $\mathcal{F}$ are the same as in Example 1 (with this assumption only p_3 may fail in any execution). Since $\forall S \in \mathcal{S}, \forall F \in \mathcal{F} : S(p_1) \cap S(p_2) \not\subseteq F$, it is easy to see that no graph has independence number higher than 2 in $\mathcal{G}_{\mathcal{Q},\mathcal{F}}$, thus the inconsistency number of $(\mathcal{Q}, \mathcal{F})$ is 2.

Computing inconsistency parameters. A straightforward approach to find the inconsistency number of $(\mathcal{Q}, \mathcal{F})$ consists in computing the independence number of all graphs $G_{F,S} \in \mathcal{G}_{\mathcal{Q},\mathcal{F}}$. The problem of finding the *maximum independent set* of a graph, and consequently its independence number, is the *maximum independent set problem* [39], known to be *NP-complete* [35]. Also, the number of graphs in $\mathcal{G}_{\mathcal{Q},\mathcal{F}}$ may exponentially grow with the number of processes. However, as the graphs might have similar structures (for example, the same quorums for some processes may appear in multiple graphs), in many practical scenarios, we should be able to avoid redundant calculations and reduce the overall computational costs, as we show for the uniform model.

Table 1 Inconsistency numbers for classical BQS with 100 processes.

Faulty processes	0–33	34–50	51–55	56–58	59–60	61	62	63	64	65	66
Inconsistency Number	1	2	3	4	5	6	7	9	12	17	34

Inconsistency in the uniform model. Centralized quorum systems generate graphs that are similar in structure and are therefore easier to analyse. Given a uniform quorum system $\mathcal{Q}_u$, we show how to calculate the inconsistency number of $(\mathcal{Q}_u, \mathcal{F}_u)$, where $\mathcal{Q}_u$ and $\mathcal{F}_u$ include every subset of processes with sizes q and $\leq f$ respectively, in which $f < q$.

▶ **Theorem 6.** *Let $(\mathcal{Q}_u, \mathcal{F}_u)$ be a uniform quorum system with n processes, quorums of size q and where at most f processes might fail. The inconsistency number of $(\mathcal{Q}_u, \mathcal{F}_u)$ is $\lfloor \frac{n-f}{q-f} \rfloor$.*

Proof. Fix any $F \in \mathcal{F}_u$ of size f. Let $G_{F,S} \in \mathcal{G}_{\mathcal{Q}_u,\mathcal{F}_u}$ be a graph whose independence number is the highest, and let $C_{max} = \{p_1, ..., p_m\}$ be a maximum independent set in $G_{F,S}$. Let $cor(Q)$ denote the number of correct processes in a quorum Q and let $Q_i = S(p_i)$. It follows that $cor(Q_1) + ... + cor(Q_m) \leq n - f$, since the quorums $Q_1, ..., Q_m$ have no correct processes in common. We can then build a graph $G_{F,S'}$ with an independent set $C' = \{p_1, ..., p_m\}$, where $\forall p_i \in C' : F \subseteq S'(p_i)$, that is, the quorum for every $p_i \in C'$ includes all faulty processes. Indeed, it suffices to choose $S'(p_i)$ as any $q - f$ correct processes from $S(p_i)$, in addition to the f faulty processes. As there can be at most $k_{max} = \lfloor \frac{n-f}{q-f} \rfloor$ disjoint sets of $q - f$ correct processes, we conclude that the maximum value m can reach is k_{max}. ◀

▶ **Example 7.** A classical *Byzantine quorum system* (BQS) uses quorums of size $q = 2n/3 + 1$. It is typically assumed that $f < n/3$ processes. As Theorem 6 implies, the inconsistency number of this range is 1, and it grows with f. Table 1 illustrates how the inconsistency number varies with the number of faulty processes in a system with 100 processes.

4 Asset Transfer System

In this section, we define the problem of k-spending asset transfer (k-AT) and describe a protocol that solves k-AT in a given trust model $(\mathcal{Q}, \mathcal{F})$, where k is the inconsistency number of $(\mathcal{Q}, \mathcal{F})$.

4.1 Preliminaries

Transactions. A transaction $tx \in \mathcal{T}$ is a tuple $(s, \tau, I, data)$, where s is the process identifier of the *issuer*, $\tau : \Pi \to \mathbb{Z}_0^+$ is the *output map* and $I \subseteq \mathcal{T}$ is the set of *input* transactions, tx is called *outgoing from* s and *incoming to* every p such that $tx.\tau(p) > 0$. Also, every transaction in $tx.I$ must be *incoming to* $tx.s$. Finally, $data$ is a bit-string attached to the transaction which contains some arbitrary information.

We use the function $inValue\colon \mathcal{T} \to \mathbb{Z}_0^+$ to denote the sum of the amount sent to s by the transaction inputs, i.e., $inValue(tx) = \sum_{tx' \in tx.I} tx'.\tau(s)$. The function $outValue\colon \mathcal{T} \to \mathbb{Z}_0^+$ denotes the total amount spent in a transaction, i.e., $outValue(tx) = \sum_{p \in \Pi} tx.\tau(p)$. A transaction tx is *valid iff* $outValue(tx) > 0$ and $outValue(tx) = inValue(tx)$. Since the issuer of tx might not send the entire value of its inputs to other processes, we allow the remaining amount to be transferred back to the issuer in its output map. We assume from this point on that each transaction in a history is signed by its issuer.

We assume that the total stake is initially distributed in a special transaction $tx_{init} = (\perp, \tau_{init}, \emptyset)$. The total stake of the system is therefore $\sum_{p \in \Pi} \tau_{init}(p)$.

Two distinct transactions tx and tx' *conflict* if they are issued by the same process and share some input, i.e., $(tx.s = tx'.s) \wedge (tx.I \cap tx'.I \neq \emptyset)$.

Transaction histories. A set of transactions $T \subseteq \mathcal{T}$ is called a *transaction history*. T generates a directed graph, where each $tx \in T$ is a node and directed edges are drawn to tx from its inputs. Let $tx, tx' \in T$, if tx is reachable from tx' in this graph (i.e., there is a path from tx' to tx), we say tx depends on tx'. A transaction history T is *well-formed iff*:

- (T-Validity) $tx_{init} \in T \wedge \forall tx \in T, tx \neq tx_{init} : tx$ is valid;
- (Completeness) $\forall tx \in T, \forall tx' \in tx.I : tx' \in T$;
- (No-Conflict) $\forall tx, tx' \in T : tx$ and tx' do not conflict;
- (Cycle-Freedom) $\forall tx, tx' \in T : tx$ depends on $tx' \Rightarrow tx \notin tx'.I$.

We only consider well-formed histories from this point on. The function $balance_T : \Pi \rightarrow \mathbb{Z}$ applied to a transaction history T determines the *balance* of each process w according to T: $balance_T(w)$ is the difference between the sum of transfers to w and the sum of transfers issued by w, i.e.,

$$balance_T(w) = \sum_{tx \in T} tx.\tau(w) \; - \sum_{tx \in T, tx.s = w} outValue(tx)$$

▶ **Proposition 8.** *Given a well-formed history T, for every process w, $balance_T(w) \geq 0$.*

Proof. Let $\sum_{tx \in T} tx.\tau(w)$ be the *incoming stake* to w and let $\sum_{tx \in T} outValue(tx)$, with $tx.s = w$, be the *outgoing stake* from w. Assume that $\text{balance}_T(w) < 0$, then the outgoing stake is greater than the incoming stake. The initial transaction tx_{init} may only send funds to w, and since every other transaction $tx \in T$ is valid, tx must include inputs with enough funds to cover *outValue(tx)*. From *Completeness*, for every transaction tx appearing in the sum of the outgoing stake, its inputs $tx' \in tx.I$ also appear in the sum of the incoming stake. Therefore, the only remaining way w can spend more stake than it received is to use an input more than once, which is prevented by *No-Conflict*. ◀

Although a well-formed history has no conflicting transactions, there may exist conflicts among distinct well-formed histories. Consider a collection of well-formed transaction histories Γ, a process r, and tx an incoming transaction to r. Let $I^r_{tx} \subseteq \mathcal{T}$ be the set of outgoing transactions from r, each $tx' \in I^r_{tx}$ including tx in its input and appearing in some $T_i \in \Gamma$,

$$I^r_{tx} = \{tx' \mid \exists T_i \in \Gamma : (tx' \in T_i) \wedge (tx \in tx'.I) \wedge (tx'.s = r)\}.$$

Let $|I^r_{tx}| = k$, we say that process r *k-spends* tx in Γ. In other words, a process k-spends if it issued k distinct transactions appearing in Γ using the same input.

▶ **Definition 9** (Spending Number). *Let Γ be a collection of well-formed histories. The* spending number *of Γ, noted $\gamma(\Gamma)$, is the highest amount of times an input is spent by the same process in Γ. Formally,*

$$\gamma(\Gamma) = max(\{|I^r_{tx}| \mid \forall r \in \Pi, \forall tx \text{ incoming to } r\}).$$

Note that, by definition, the spending number of Γ cannot exceed $|\Gamma|$.

4.2 Problem Statement

Every process $p \in \Pi$ maintains a *local history* T_p, where p *accepts* tx when it adds tx to T_p.

Ideally, we want local histories of correct processes to eventually converge. But this may not always be possible, as our specification allows for multiple spending: correct process may accept conflicting transactions. Therefore, we also introduce an accountability mechanism, expressed in the form of accusation histories.

Formally, an *accusation* is a tuple (AC, P) consisting of a set of processes $AC \subseteq \Pi$ and a *proof of misbehavior* P for every process in AC. (AC, P) can be independently verified by a third party through the function *verify-acc:* $(2^{\Pi} \times \mathcal{P}) \rightarrow \{true, false\}$. Technically, for each process $p \in AC$, the proof P must contain a set of conflicting transactions $tx_1, ..., tx_\ell$ signed by p. We say that the accusation (AC, P) *refers* to $tx_1, ..., tx_\ell$.

Every process p is also expected to maintain a local accusation history A_p, where each element in A_p is an accusation tuple. The *k-spending asset transfer abstraction* receives inputs of the form *transfer(tx)* and produces updates to the local histories T_p and A_p.

Consider a run of a *k-spending asset transfer protocol* (k-AT) in a trust model $(\mathcal{Q}, \mathcal{F})$ with a fixed faulty set $F \in \mathcal{F}$. Let $T_p(t)$ and $A_p(t)$ denote the transaction history and accusation history of process p at time t, respectively. Let $\Gamma(t)$ denote the collection of local histories of correct processes at time t. Then the run must satisfy:

Validity If a correct process issues a transaction tx, then every live correct process p eventually adds tx to T_p, or adds an accusation to A_p referring to some transaction on which tx depends.

k-Spending For all $t \geq 0$, the spending number of $\Gamma(t)$ is bounded by k, i.e., $\gamma(\Gamma(t)) \leq k$.

Eventual Conviction If correct processes p and q add conflicting transactions tx to T_p and tx' to T_q respectively, then they eventually add an accusation referring to tx and an accusation referring to tx' to A_p and A_q.

Accuracy For all $t \geq 0$ and (AC, P) in $A_p(t)$: *verify-acc*$(AC, P) =$ *true*. Moreover, *verify-acc*(AC, P) returns *true* if and only if $AC \subseteq F$.

Agreement If a correct process p adds an accusation (AC, P) to A_p, then every correct process eventually adds (AC, P) to its accusation history.

Integrity If $tx.s$ is correct, a correct process p adds tx to T_p only if $tx.s$ previously issued tx.

Monotonicity The accusation history of correct processes grows monotonically, i.e., for all p correct and $t \leq t'$, $A_p(t) \subseteq A_p(t')$;

Termination If a correct process p adds a transaction tx to T_p, then every live correct process q eventually adds tx to T_q or an accusation referring to tx (or some transaction on which tx depends) to A_q.

4.3 k-Spending Asset Transfer Protocol

The pseudo-code of our *k-spending asset transfer protocol* is presented in Algorithms 1 and 2. In the protocol, a process accepts a transaction only after hearing from a (local) quorum, and after all of the transaction's inputs have already been accepted.

Local Variables. Variables *echoes*, *usedInp* and *pending* are used in a broadcast stage of the algorithm. The array *echoes* stores received transactions echoed by other processes. In *usedInp*, p_i stores all transactions it has witnessed to be used as inputs, while in *pending* it stores transactions with signatures from at least a quorum that have not yet been added to the history. The remaining variables are: p_i's transaction history *trHist*, p_i's accusation history *acHist*, and *signedReq*, an array with sets of tuples (tx, σ), where σ is a signature for tx from $tx.s$.

The complete algorithm consists of three main blocks: the *broadcast* block, the *acceptance* block and the *accountability* block. In the following, we give a detailed description on how each block operates.

Broadcasting transactions. In order to issue a transaction, p_i specifies a transaction tx and invokes the operation $transfer(tx)$ (we assume that transactions issued by correct processes are always valid). Process p_i then creates a signature σ for tx and sends them in a REQ message to every process in the system. Upon receiving REQ with tx, p_i stores the signed transaction in *signedReq*. If none of tx's inputs are in $usedInp[tx.s]$, p_i echoes the original signed request with the issuer's signature and adds the inputs of the transaction to $usedInp[tx.s]$. A message whose signature does not match its sender is ignored.

Each time a new $ECHO$ is received from p_j for a transaction tx, p_i stores the echoed transaction in $echoes[p_j]$ and follows the same steps as when receiving a REQ message. When "enough" echoes are collected for the same transaction tx, and if tx is neither in *pending* nor *trHist*, it is added to *pending*.

Accepting transactions. After going through the broadcast phase and adding tx to *pending*, the function $ready(tx)$ is used to verify whether the addition of tx to *trHist* results in a well-formed history. If *T-Validity*, *Completeness*, and *No-Conflict* still hold, p_i adds tx to *trHist* and removes it from *pending* (as later shown in Lemma 10, Cycle-Freedom is guaranteed by construction since a single transaction is added at a time).

Treating accusations. Since p_i keeps track of every received request (tx, σ) in *signedReq* (either coming directly from a REQ message or from an $ECHO$), it can construct a proof of misbehavior after receiving signed conflicting transactions. The proof here consists of a pair (tx, σ_j) and (tx', σ'_j) containing distinct transactions from p_j whose inputs have a non-empty intersection. An accusation (AC, P) is created using p_j's identifier and the proof. If it is a new accusation, p_i adds (AC, P) to *acHist* and sends it to every process in an ACC message. The same steps are followed once an ACC is received with a verifiable accusation tuple (AC, P).

Correctness. Consider executions of Algorithms 1 and 2 assuming trust model $(\mathcal{Q}, \mathcal{F})$ with inconsistency number k_{max}. Let $F \in \mathcal{F}$ be a corresponding faulty set.

▶ **Lemma 10.** *The history T_p of a correct process p is well-formed, i.e., satisfies the properties of T-Validity, Completeness, No-Conflict and Cycle-Freedom.*

Proof. The default value of *trHist* is $\{tx_{init}\}$, which is well-formed by definition. Now assume that at some point *trHist* is well-formed. Before adding a new transaction tx at line 21, p invokes $ready(tx)$ to check whether tx is valid and that the resulting history satisfies *No-Conflict* and *Completeness* (lines 34 to 38). By construction, *trHist* is also *Cycle-Free*: suppose $\{tx\} \cup trHist$ creates a cycle, that is, $\exists tx' \in \{tx\} \cup trHist$ on which tx depends and $tx \in tx'.I$. This is clearly not possible: since *trHist* satisfies *Completeness*: $\forall tx'' \in tx'.I : tx'' \in trHist$, but $tx \notin trHist$, a contradiction. ◀

▶ **Lemma 11** (k-Spending)**.** *At any time t, the spending number of $\Gamma(t)$ is bounded by k_{max}.*

Proof. Let $r \in F$ and tx be an incoming transaction to r. Suppose r spends tx k times in $\Gamma(t)$, with $k > k_{max}$. We assume, without loss of generality, that r is the process that multiple spent the maximal number of times in $\Gamma(t)$, that is, $\gamma(\Gamma(t)) = k$. We make the following observations about the algorithm:

Algorithm 1 k-Spending Asset Transfer System: code for process p_i part 1.

Local Variables:
$echoes \leftarrow [\emptyset]^N$; `/* Array containing sets of received echoes */`
$usedInp \leftarrow [\emptyset]^N$; `/* Array of inputs used by each process */`
$pending \leftarrow \emptyset$; `/* Set of transactions waiting to be accepted */`
$trHist \leftarrow \{tx_{init}\}$; `/*` Transaction History of p_i `*/`
$signedReq \leftarrow [\emptyset]^N$; `/* An array of set of pairs transaction-signature */`
$acHist \leftarrow \emptyset$; `/*` Accusation history of p_i `*/`

behavior:
Ignore messages with invalid signatures;

operation $transfer(tx)$**:**
  $\sigma \leftarrow sign(self, tx)$;
  send $\langle REQ, tx, \sigma \rangle$ to all $p \in \Pi$;

upon *receiving* $\langle REQ, tx, \sigma_j \rangle$ *from* p_j**:**
  $signedReq[tx.s] \leftarrow signedReq[tx.s] \cup \{(tx, \sigma_j)\}$;
  if $tx.I \cap usedInp[tx.s] = \emptyset$ **then:**
    $usedInp[tx.s] \leftarrow usedInp[tx.s] \cup tx.I$; `/* Stores used inputs */`
    $\sigma \leftarrow sign(self, tx)$;
    send message $\langle ECHO, (tx, \sigma_j), \sigma \rangle$ to all $p \in \Pi$;

upon *receiving* $\langle ECHO, (tx, \sigma_s), \sigma_j \rangle$ *from* p_j**:**
  $echoes[p_j] \leftarrow echoes[p_j] \cup \{tx\}$;
  $signedReq[tx.s] \leftarrow signedReq[tx.s] \cup \{(tx, \sigma_s)\}$;
  if $tx.I \cap usedInp[tx.s] = \emptyset$ **then:**
    $usedInp[tx.s] \leftarrow usedInp[tx.s] \cup tx.I$;
    $\sigma \leftarrow sign(self, tx)$;
    send message $\langle ECHO, (tx, \sigma_s), \sigma \rangle$ to all $p \in \Pi$;

upon *receiving echoes for* tx *from a quorum* $Q_i \in \mathcal{Q}(p_i)$**:**
  if $tx \notin trHist \wedge tx \notin pending$ **then:**
    $pending \leftarrow pending \cup \{tx\}$; `/*` Collected enough signatures for tx `*/`

Algorithm 2 k-Spending Asset Transfer System: code for process p_i part 2.

upon *existing* $tx \in pending$ *such that* $ready(tx) = \textbf{\textit{true}}$ **:**
  $trHist \leftarrow trHist \cup \{tx\}$; `/* Adds transaction to history */`
  $pending \leftarrow pending / \{tx\}$;

upon *existing distinct* tx *and* tx' *in* $signedReq[p_j]$ *such that* $tx.I \cap tx'.I \neq \emptyset$**:**
  $ev_1 \leftarrow (tx, \sigma_j)$; `/* Evidences of misbehavior */`
  $ev_2 \leftarrow (tx', \sigma'_j)$;
  $accusation \leftarrow (\{p_j\}, \{ev_1, ev_2\})$; `/*` $AC = \{p_j\}$, $P = \{ev_1, ev_2\}$ `*/`
  if $accusation \notin acHist$ **then:**
    $acHist \leftarrow acHist \cup \{accusation\}$; `/* Adds accusation to history */`
    send $\langle ACC, accusation \rangle$ to all $p \in \Pi$;

upon *receiving* $\langle ACC, accusation \rangle$ *from* p_j**:**
  if $accusation \notin acHist \wedge verify\text{-}acc(accusation)$ **then:**
    $acHist \leftarrow acHist \cup \{accusation\}$;
    send $\langle ACC, accusation \rangle$ to all $p \in \Pi$;

operation $ready(tx)$**:**
  $c_1 \leftarrow \forall tx' \in tx.I : tx' \in trHist$; `/*` *Completeness* `*/`
  $c_2 \leftarrow$ **true** *iff* tx is valid; `/*` *T-Validity* `*/`
  $c_3 \leftarrow \forall tx' \in trHist : (tx'.s = tx.s) \Rightarrow (tx'.I \cap tx.I = \emptyset)$; `/*` *No-Conflict* `*/`
  return $c_1 \wedge c_2 \wedge c_3$;

1. A correct process p adds a transaction tx' to its history only if it received *ECHO* messages for tx' from every process in a quorum $Q \in \mathcal{Q}(p)$ (guard in line 17).
2. p checks if any input of a received transaction is already in *usedInp* before echoing it (lines 6 and 13), and if it sends *ECHO* for a transaction, p adds all of its inputs to *usedInp* (lines 7 and 14). Therefore, p can only send *ECHO* for a single transaction from r that has tx as an input.

Let correct processes p_i and p_j accept conflicting transactions tx_i and tx_j from r after receiving echoes from $Q_i \in \mathcal{Q}(p_i)$ and $Q_j \in \mathcal{Q}(p_j)$, respectively. From (2) above, we conclude that $Q_i \cap Q_j \subseteq F$, otherwise a correct process in the intersection would have echoed two different transactions sharing some input(s) from r.

Since r k-spends tx in $\Gamma(t)$, there exists $p_1, ..., p_k$ correct that accepted, respectively, conflicting $tx'_1, ..., tx'_k$ from r using tx as input. Now let $Q_1 \in \mathcal{Q}(p_1), ..., Q_k \in \mathcal{Q}(p_k)$ be the quorums each process received echoes from before accepting the transactions. We can build a quorum map S satisfying $S(p_i) = Q_i$ for $i = 1, ..., k$, and a graph $G_{F,S} \in \mathcal{G}_{\mathcal{Q},\mathcal{F}}$ of which $C = \{p_1, ..., p_k\}$ is an independent set, since from (1) and (2) above: $\forall p_i, p_j \in C, i \neq j : S(p_i) \cap S(p_j) \subseteq F$. However, k_{max} is the inconsistency number of $(\mathcal{Q}, \mathcal{F})$, which means there cannot be a graph $G_{F,S} \in \mathcal{G}_{\mathcal{Q},\mathcal{F}}$ with an independent set of size $k > k_{max}$, a contradiction. ◀

▶ **Lemma 12** (Eventual Conviction). *If correct processes p and q add conflicting transactions tx to T_p and tx' to T_q, respectively, then they eventually add an accusation referring to tx and an accusation referring to tx' to A_p and A_q respectively.*

Proof. Before accepting tx and tx', p and q received echoes for tx (in p's case) and tx' (in q's case), storing the original signed requests in their local *signedReq* (line 12). There are two possible scenarios for each process (for simplicity, we only describe them for p): p echoed tx before adding it to T_p, or p did not echo tx. If p echoed tx, then q will eventually receive the echo with a signed request for tx from p, which allows q to construct and relay an accusation for $tx.s$ (in lines 23 to 29) using this request together with the one for tx' already stored in q's *signedReq* (e.g. assigning the request for tx to ev_1 at line 24 and the request for tx' to ev_2 at line 25). Eventually p will receive an *ACC* message from q containing this accusation and will add it to its accusation history.

Now if p did not echo tx, then it must have echoed for another conflicting transaction tx'', which means p can construct an accusation using the respective signed requests for tx and tx'' as described above. This accusation is sent to every process and is eventually received by q, which then adds it to *acHist*. Ultimately, both p and q add accusations referring to tx and tx' to their accusation histories. ◀

▶ **Lemma 13** (Termination). *If a correct process p adds a transaction tx to T_p, then every live correct process q eventually adds tx to T_q or an accusation referring to tx (or some transaction on which tx depends) to A_q.*

Proof. Recall that a process is live if it has a quorum composed of only correct processes.

We first show the following: If a correct process adds a transaction tx to *pending*, then every live correct process eventually does so or adds an accusation referring to tx to *acHist*.

Let p be a correct process that adds tx to its *pending* after receiving echoes for tx from a quorum. There are two cases to consider, depending on whether p previously echoed tx or not.

If p did not echo tx, then it echoed a conflicting tx' and built an accusation (AC, P) with the original requests for tx and tx' (lines 23 to 26). Then, p adds the accusation to its *acHist* and sends (AC, P) to all processes. Every correct process eventually receives the accusation and also adds it to *acHist*.

Suppose now that p echoed *tx*. If no process sent *ECHO* or *REQ* for a conflicting transaction, then every correct process eventually receives and echoes *tx*. If a correct process q is live, it will eventually receive enough echoes and add *tx* to *pending*. On the other hand, if a process in q's live quorum had echoed a conflicting transaction, q will receive the conflicting requests, build an accusation (AC, P) referring to *tx* and tx' and send it to all processes. Then, as described previously, every correct process will eventually add (AC, P) to *acHist*.

Now suppose p also adds *tx* to its *trHist*. We make the following observations about the algorithm: before being added *trHist*, any transaction tx' is first added to *pending* (guard in line 20). Also, from Lemma 10 every transaction on which tx' depends must have been previously added to *trHist*. Let *deps(tx)* include *tx* and every transaction on which *tx* depends. It follows that p previously added every $tx' \in$ *deps(tx)* to *pending*. The following three cases are then possible for a live correct process q:

1. q eventually adds every $tx' \in$ *deps(tx)* to its *pending*. If no transaction in *trHist* conflicts with them, q adds every such tx' to *trHist*.
2. q has already added a transaction to *trHist* that conflicts with some $tx' \in$ *deps(tx)*. In this case, it received conflicting requests. q will then build and send everybody an accusation including the signed requests for the respective transactions.
3. q never adds one (or more) $tx' \in$ *deps(tx)* to *pending*, in which case, as previously shown, q eventually adds an accusation referring to tx' to *acHist*.

Therefore, if a correct process p adds a transaction *tx* to *trHist* and a live correct process q is never able to do so, then q eventually adds an accusation to *acHist* referring to *tx* or some transaction on which *tx* depends. ◀

▶ **Lemma 14** (Validity). *If a correct process issues a transaction tx, then every live correct process p eventually adds tx to* T_p*, or adds an accusation to* A_p *referring to some transaction on which tx depends.*

Proof. If correct process p sends a request for *tx*, eventually every correct process echoes *tx* and every live correct process adds *tx* to its *pending*. Since p is correct, it will not send conflicting requests, thus no accusation referring to *tx* can be produced. Also, p must have previously added every transaction on which *tx* depends to *trHist*, which from Lemma 13, if a live correct process q does not add said transactions to *trHist* (and consequently *tx*), q eventually adds an accusation to *acHist* referring to some transaction on which *tx* depends. ◀

▶ **Theorem 15.** *Consider the trust model* $(\mathcal{Q}, \mathcal{F})$ *with inconsistency number* k_{max}*. Algorithms 1 and 2 implement* k_{max}*-spending asset transfer abstraction.*

Proof. Lemma 10 shows that correct processes always maintains well-formed local transaction histories. The k_{max}*-Spending*, *Eventual Conviction*, *Termination* and *Validity* properties are shown in Lemmas 11 to 14.

Accuracy, *Monotonicity*, *Agreement* and *Integrity* are immediate. A correct process adds an accusation (AC, P) to *acHist* only if it can verify that messages for conflicting transactions in P were indeed signed by the processes in AC (*Accuracy*). The set *acHist* may only grow with time (*Monotonicity*). Moreover, once a correct process adds an accusation to its *acHist*, it sends the accusation to every other process. This accusation is eventually received by every correct process, which verifies and adds it to *acHist* (*Agreement*). Finally, the signature of a correct process for a transaction request cannot be forged (*Integrity*). ◀

5 Relaxed Broadcast Abstraction and Lower Bounds

In this section, we show that the inconsistency number of $(\mathcal{Q}, \mathcal{F})$ is optimal for k-AT, by relating the problem to the fundamental *broadcast abstraction*. The abstraction exports one operation *broadcast(m)* and enables a callback *deliver(m)*, for m in a value set $\mathcal{M}$. We assume that each broadcast instance has a dedicated *source*, i.e., the process invoking the broadcast operation.

We now describe *k-Consistent Broadcast*, first introduced in [5]. Given a trust model $(\mathcal{Q}, \mathcal{F})$, in every execution with a fixed $F \in \mathcal{F}$, a k-Consistent Broadcast protocol ensures the following properties:

- (Validity) If the source is correct and broadcasts m, then every *live* correct process eventually delivers m.
- (k-Consistency) Let M be the set of values delivered by correct processes, then $|M| \leq k$.
- (Integrity) A correct process delivers at most one value and, if the source p is correct, only if p previously broadcast it.

This protocol is a generalized version of an abstraction known as *Consistent Broadcast* [9]. *Validity* in Consistent Broadcast guarantees that a broadcast value is delivered by every correct process. Also, correct processes cannot deliver different values. Note that if every correct process is live and $k = 1$, then k-CB implements Consistent Broadcast.

5.1 Lower bound for k-Consistent Broadcast

We restrict our attention to *quorum-based protocols*, initially introduced in the context of consensus algorithms [32]. Intuitively, in a quorum-based protocol, a process p should be able to make progress if the members in one of its quorums $Q \in \mathcal{Q}$ appear correct to p. This should hold even if the actual set of correct processes in this execution is different from Q. Formally, we make the following assumption about algorithms implementing k-CB:

- (Local Progress) For all $p \in \Pi$ and $Q \in \mathcal{Q}(p)$, there is an execution in which only the source and processes in Q take steps, p is correct, and p delivers a value.

Consider a trust model $(\mathcal{Q}, \mathcal{F})$ and let k_{max} be its inconsistency number, then:

▶ **Theorem 16.** *No algorithm can implement k-CB such that $k < k_{max}$.*

Proof. Let $G_{F,S}$ be the graph generated over fixed $F \in \mathcal{F}$ and $S \in \mathcal{S}$ and $C = \{p_1, ..., p_{k'}\}$ an independent set in $G_{F,S}$ of size k'. We proceed to show that there exists an execution where k' different values are delivered by processes in C. Let r be the source, by the definition of *Local Progress*, there exists an execution Σ_i for each $p_i \in C$ where $part(\Sigma_i) = \{r\} \cup S(p_i)$, in which p_i delivers a value m_i. Now assume that r is faulty, we can build an execution Σ such that all the correct processes in $part(\Sigma_i)$ take the same steps in Σ as in Σ_i, and all the faulty processes behave to them (send the same messages) the same as in execution Σ_i. For a correct process in $part(\Sigma_i)$, Σ is then indistinguishable from Σ_i, and thus $p_i, ..., p_{k'}$ must deliver $m_1, ..., m_{k'}$ respectively.

Now let $G'_{F,S} \in \mathcal{G}_{\mathcal{Q},\mathcal{F}}$ be a graph whose independence number is k_{max}, there exists an independent set C_{max} of size k_{max} in $G'_{F,S}$. As shown above, it is possible to build an execution where k_{max} processes ($k_{max} \geq k'$) deliver k_{max} distinct values before any correct process is able to identify the misbehavior. ◀

Intuitively, if two correct processes have quorums that do not have a correct process in the intersection, they might deliver distinct values before noticing any misbehavior in the execution. Within an independent set, the quorums of every pair of nodes do not intersect in a correct process, and k_{max} represents the highest possible independent set in $\mathcal{G}_{\mathcal{Q},\mathcal{F}}$, thus establishing the lower bound for k-CB.

5.2 Relating k-Spending Asset Transfer and k-Consistent Broadcast

We show now that having a protocol implementing k-AT, one implement k-CB, which implies that the lower bound established in Theorem 16 also holds for k-AT.

▶ **Theorem 17.** *k-AT can be used to implement k-CB.*

Proof. Suppose that we have a protocol implementing k-AT. First, we let tx_{init} assign some funds to the source p, and assume that p broadcasts a message to other processes by means of encoding it in a transaction's *data*.

Therefore, to broadcast a message m, p issues a transaction $tx = (p, \tau, \{tx_{init}\}, m)$. Whenever a correct process q adds tx to T_q, it issues *deliver(m)*.

If p is correct, every live correct process eventually delivers it, that is k-AT *Validity* implies k-CB *Validity*. Moreover, since processes do not have knowledge of $\mathcal{F}$, an algorithm implementing k-AT must guarantee *Validity* for an arbitrary fault scenario $F \in \mathcal{F}$. As such, for particular source p, process p_i and quorum $Q_i \in \mathcal{Q}(p_i)$, if F is such that every process other than $p \cup Q_i$ is faulty, then there must be an execution in which only these processes take step and p_i accepts the transaction from p, implying the *Local Progress* assumed in k-CB protocols.

From the *k-Spending* property, up to k conflicting transactions issued by p with tx_{init} as input can be accepted by correct processes. Thus, at most k distinct messages might be "delivered", which implies *k-Consistency*. Trivially, k-AT *Integrity* implies k-CB *Integrity*. ◀

Theorems 16 and 17 imply that Algorithms 1 and 2 implement k-AT with optimal k.

6 Related Work

Damgård et al. [16] appear to be the first to consider the decentralized trust setting. They introduced the notion of *aggregate adversary structure* $\mathcal{A}$: each node is assigned with a collection of subsets of nodes that the adversary might corrupt at once. In this model, assuming *synchronous* communication, they discuss solutions for broadcast, verifiable secret sharing and multiparty computation.

Ripple [37] and Stellar [34], conceived as *open* payment systems, use decentralized trust as an alternative to *proof-of-work*-based protocols [36, 41]. In the Ripple protocol, each participant expresses its trust assumptions in the form of a *unique node list* (UNL), a subset of system members. To accept a transaction, a node needs acknowledgement from a set of at least 80% of its UNL (which can be seen as a quorum). The Ripple white paper [37] assumes that up to 20% of members in a UNL are Byzantine, stating that an overlap of at least 20% between every pair of UNLs in enough to prevent *forks*. Later analyses suggest this overlap to be more than 40% [3] without Byzantine faults, and more than 90% with the same original assumptions [13] (up to 20% Byzantine members in a UNL). Chase and MacBrough [13] also provide an example in which liveness of the protocol is violated even with 99% of overlap.

Stellar consensus protocol [34] uses a *Federated Byzantine Quorum System* (FBQS). A quorum Q in the FQBS is a set that includes a *quorum slice* (a trusted subset of members) for every member in Q. Correctness of Stellar depends on the individual trust assumptions, and stronger properties are guaranteed for nodes trusting the "right guys", which are in so called *intact sets*. García-Pérez and Gotsman [19] treated Stellar consensus formally, by relating it to Bracha's Broadcast Protocol [6], build on top of a FBQS. Their analysis has been later extended [20] to a variant of state-machine replication protocol that allows *forks*, where disjoint intact sets may maintain different copies of the system state.

Losa et al. [32], Sheff et al. [38], and Cachin [8] investigate more general formalizations of decentralized trust. Cachin and Tackmann [11] model trust assumptions via an *asymmetric fail-prone system*, an array $[\mathcal{F}_1, ..., \mathcal{F}_n]$ of adversary structures (or fault models), where each $\mathcal{F}_i$ is chosen by p_i as its local fault model. For this model, they devise an *asymmetric Byzantine quorum system* (ABQS), an array of quorum systems $[\mathcal{Q}_1, ..., \mathcal{Q}_n]$ satisfying specific intersection and availability properties, so that certain problems, such as broadcast and storage, can be solved. Recently, Losa et al. [10] extended this formalism to *permissionless* settings where the processes may make assumptions about each others' trust models.

Losa et al. [32] introduced the notion of a *Personal Byzantine Quorum System* (PBQS), where every process chooses its quorums with the restriction that if Q is a quorum for a process p, then Q includes a quorum for every process $q' \in Q$. The PBQS model is then discussed in relations to Stellar consensus [34]. More precisely, they characterize the conditions on PBQS under which a *quorum-based* protocol (captured by our Local Progress condition) ensures that a well-defined subset of processes (a *consensus cluster*) can maintain safety and liveness of consensus.

In contrast, we allow the processes to directly choose their quorums, and we address the question of what is the "best" consistency a cryptocurrency can achieve within this trust model. The measure of consistency is quantified here as the spending number. In a way, unlike this prior work on decentralized trust, instead of searching for the weakest trust model that enables solutions to a given problem, we determine the "strongest" problem (in a specific class) that is possible to solve in a given model.

In the context of distributed systems, accountability has been proposed as a mechanism to detect "observable" deviations of system nodes from the algorithms they are assigned with [25, 24, 26]. Recent proposals [14, 17] focus on *application-specific* accountability that only heads for detecting misbehavior that affects correctness of the problem to be solved, e.g., consensus [14] or lattice agreement [17]. Our k-AT algorithm generally follows this approach, except that it implements a relaxed form of asset transfer system, but detects violations that affect correctness of the stronger, conventional asset transfer abstraction [22].

7 Discussion and Future Work

Generalizing the inconsistency measure. Our notion of the inconsistency number of a trust model $(\mathcal{Q}, \mathcal{F})$ serves to quantify the amount of times a process can spend the same input in our cryptocurrency implementation (or the number of distinct values that can be delivered by correct processes in our broadcast abstraction). A natural variation of this question is to explore the conditions on a trust model that are necessary and sufficient to implement a cryptocurrency which bounds the number of copies the same asset can maintain in a system. Note that our notion of k-spending is more relaxed, as it only bounds the number of times the same input transaction can be used by a process.

It is very interesting to apply "inconsistency metrics" for solving other, more general problems in the decentralized trust setting. Consider for example *State Machine Replication* (SMR) protocols [31, 12]. In these protocols, correct processes agree on a global history of concurrently applied operations, and thus witness the same evolution of the system state.

One way to relax consistency guarantees of SMR protocols in decentralized trust settings is to bound the number k of diverging histories (the maximum degree of the fork). The question is then how to relate k to the trust model $(\mathcal{Q}, \mathcal{F})$.

Reconfiguration of inconsistent states. Our k-AT abstraction provides the application with the history of valid transactions and a record of misbehaving parties. An important question is left open: once correct processes accept conflicting transactions and accusations against Byzantine processes are raised, what is next? Is there a way to render the system back to a consistent state? Although there is no general answer to these questions – it comes down to what better suits the application – we point out some of the suitable strategies.

A natural response to this is to *reconfigure* both the trust model and the states of the processes, in order to achieve some desired level of consistency. The immediate use of an accusation (AC, P) is to rearrange the trust assumptions by evicting the misbehaving parties AC from the system. For example, the application might use the accusation history to suggest new (improved) quorum systems to system members. One may hope to deploy recently developed asynchronous reconfiguration schemes [1, 29, 30].

When it comes to reconfiguration of the system states, we face a more challenging task. Indeed, some correct processes may have already used "compromised" (multiply spent) assets in their transactions. "Merging" conflicting histories into a consistent global state might affect the stake distribution, which can be hard to resolve without changing the application semantics. We present two strategies that make use of the transactions identified in accusation proofs. The first approach, alluding to real financial systems, is to let them keep (and use as input) accepted conflicting transactions after the misbehaving parties are excluded from the system. This can have implications on the total stake of the system: depending on how much stake was spent, the total system stake may grow. Once multi-spending party may get negative balance in its accounts, and could be "frozen", i.e., forbidden to issue new transactions until the balance turns positive again (due to incoming transactions). The advantage of this approach is that the system remains live despite conflicting transactions. As a downside, a malicious party might be able to spend its entire balance k times. This problem appears inevitable in asynchronous networks, and additional assumptions might be necessary if we want to have a better " overspending bound."

The second, and probably the most straightforward approach, is to *rollback* any transaction tx appearing in a proof, i.e., removing tx and every transaction depending on tx from transaction histories. Surely, this comes with the downside of invalidating a previously accepted transaction, which might affect correct system members in real life. As an alternative way of compensating correct processes in this case, the application might opt to redistribute the stake from the misbehaving parties among the harmed members.

Given a strategy for the reconfiguration of system members and states, an interesting course to follow would be in self-reconfigurable systems [17]: the protocol automatically rearrange trust assumptions and merge conflicting histories to keep the system live.

Composition of trust. Alpos et al. [2] show how to compose trust models of different (possibly disjoint) systems. Given two asymmetric fail-prone systems $[\mathcal{F}_1, ..., \mathcal{F}_n]$ and $[\mathcal{F}'_1, ..., \mathcal{F}'_m]$ and matching decentralized quorum systems, a *composed* trust model can be constructed. In the context of our relaxed cryptocurrency protocols, it is appealing to understand how the spending number of a composition of two independent systems may depend on the spending number of its components.

References

1 Marcos Kawazoe Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. Dynamic atomic storage without consensus. *J. ACM*, 58(2):7:1–7:32, 2011. doi:10.1145/1944345.1944348.

2 Orestis Alpos, Christian Cachin, and Luca Zanolini. How to trust strangers: Composition of byzantine quorum systems. In *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*, pages 120–131. IEEE, 2021. doi:10.1109/SRDS53918.2021.00021.

3 Frederik Armknecht, Ghassan O Karame, Avikarsha Mandal, Franck Youssef, and Erik Zenner. Ripple: Overview and outlook. In *International Conference on Trust and Trustworthy Computing*, pages 163–180. Springer, 2015. doi:10.1007/978-3-319-22846-4_10.

4 Mathieu Baudet, George Danezis, and Alberto Sonnino. Fastpay: High-performance byzantine fault tolerant settlement. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, pages 163–177, 2020. doi:10.1145/3419614.3423249.

5 João Paulo Bezerra, Petr Kuznetsov, and Alice Koroleva. Relaxed reliable broadcast for decentralized trust. In *International Conference on Networked Systems*, 2022.

6 Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987. doi:10.1016/0890-5401(87)90054-X.

7 Gabriel Bracha and Sam Toueg. Resilient consensus protocols. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 12–26, 1983. doi:10.1145/800221.806706.

8 Christian Cachin. Asymmetric distributed trust. In *International Conference on Distributed Computing and Networking 2021*, pages 3–3, 2021. doi:10.1145/3427796.3433933.

9 Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011. doi:10.1007/978-3-642-15260-3.

10 Christian Cachin, Giuliano Losa, and Luca Zanolini. Quorum systems in permissionless networks. In *OPODIS*, 2022. doi:10.4230/LIPICS.OPODIS.2022.17.

11 Christian Cachin and Björn Tackmann. Asymmetric distributed trust. In *OPODIS*, volume 153, pages 7:1–7:16, 2019. doi:10.4230/LIPICS.OPODIS.2019.7.

12 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *OSDI: Symposium on Operating Systems Design and Implementation*. USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS, feb 1999.

13 Brad Chase and Ethan MacBrough. Analysis of the xrp ledger consensus protocol. *arXiv preprint*, 2018. arXiv:1802.07242.

14 Pierre Civit, Seth Gilbert, and Vincent Gramoli. Polygraph: Accountable byzantine agreement. *IACR Cryptol. ePrint Arch.*, 2019:587, 2019. URL: https://eprint.iacr.org/2019/587.

15 Daniel Collins, Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Yvonne Anne Pignolet, Dragos-Adrian Seredinschi, Andrei Tonkikh, and Athanasios Xygkis. Online payments by merely broadcasting messages. In *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2020, Valencia, Spain, June 29 - July 2, 2020*, pages 26–38. IEEE, 2020. doi:10.1109/DSN48063.2020.00023.

16 Ivan Damgård, Yvo Desmedt, Matthias Fitzi, and Jesper Buus Nielsen. Secure protocols with asymmetric trust. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 357–375. Springer, 2007. doi:10.1007/978-3-540-76900-2_22.

17 Luciano Freitas de Souza, Petr Kuznetsov, Thibault Rieutord, and Sara Tucci Piergiovanni. Accountability and reconfiguration: Self-healing lattice agreement. *CoRR*, abs/2105.04909, 2021. arXiv:2105.04909.

18 Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985. doi:10.1145/3149.214121.

19 Álvaro García-Pérez and Alexey Gotsman. Federated byzantine quorum systems (extended version). *arXiv preprint*, 2018. arXiv:1811.03642.

20 Álvaro García-Pérez and Maria A Schett. Deconstructing stellar consensus (extended version). *arXiv preprint*, 2019. arXiv:1911.05145.

21 Guy Golan-Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K. Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. SBFT: A scalable and decentralized trust infrastructure. In *DSN*, pages 568–580. IEEE, 2019. doi:10.1109/DSN.2019.00063.

22 Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. The consensus number of a cryptocurrency. In Peter Robinson and Faith Ellen, editors, *PODC*, pages 307–316. ACM, 2019. doi:10.1145/3293611.3331589.

23 Saurabh Gupta. *A non-consensus based decentralized financial transaction processing model with support for efficient auditing.* Arizona State University, 2016.

24 Andreas Haeberlen and Petr Kuznetsov. The Fault Detection Problem. In *Proceedings of the 13th International Conference on Principles of Distributed Systems (OPODIS'09)*, dec 2009. doi:10.1007/978-3-642-10877-8_10.

25 Andreas Haeberlen, Petr Kuznetsov, and Peter Druschel. The case for byzantine fault detection. In *Proceedings of the Second Workshop on Hot Topics in System Dependability (HotDep'06)*, nov 2006. URL: https://www.usenix.org/conference/hotdep-06/case-byzantine-fault-detection.

26 Andreas Haeberlen, Petr Kuznetsov, and Peter Druschel. PeerReview: Practical accountability for distributed systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, oct 2007. doi:10.1145/1294261.1294279.

27 Jonathan Katz and Chiu-Yuen Koo. On expected constant-round protocols for byzantine agreement. In *Annual International Cryptology Conference*, pages 445–462. Springer, 2006. doi:10.1007/11818175_27.

28 Ramakrishna Kotla, Lorenzo Alvisi, Michael Dahlin, Allen Clement, and Edmund L. Wong. Zyzzyva: Speculative byzantine fault tolerance. *ACM Trans. Comput. Syst.*, 27(4):7:1–7:39, 2009. doi:10.1145/1658357.1658358.

29 Petr Kuznetsov, Thibault Rieutord, and Sara Tucci Piergiovanni. Reconfigurable lattice agreement and applications. In *OPODIS*, volume 153 of *LIPIcs*, pages 31:1–31:17, 2019. doi:10.4230/LIPICS.OPODIS.2019.31.

30 Petr Kuznetsov and Andrei Tonkikh. Asynchronous reconfiguration with byzantine failures. In Hagit Attiya, editor, *DISC*, volume 179 of *LIPIcs*, pages 27:1–27:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICS.DISC.2020.27.

31 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, jul 1978. doi:10.1145/359545.359563.

32 Giuliano Losa, Eli Gafni, and David Mazières. Stellar consensus by instantiation. In *33rd International Symposium on Distributed Computing (DISC 2019)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICS.DISC.2019.27.

33 Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed computing*, 11(4):203–213, 1998. doi:10.1007/S004460050050.

34 David Mazieres. The stellar consensus protocol: A federated model for internet-level consensus. *Stellar Development Foundation*, 32, 2015.

35 R Miller. *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations, held March 20 22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and sponsored by the Office of Naval Research, Mathematics Program, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department.* Springer Science & Business Media, 2013. doi:10.1007/978-1-4684-2001-2.

36 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, page 21260, 2008.

37 David Schwartz, Noah Youngs, Arthur Britto, et al. The ripple protocol consensus algorithm. *Ripple Labs Inc White Paper*, 5(8):151, 2014.

38 Isaac C. Sheff, Xinwen Wang, Robbert van Renesse, and Andrew C. Myers. Heterogeneous paxos: Technical report. In *OPODIS*, 2020. doi:10.4230/LIPIcs.OPODIS.2020.5.

39 Robert Endre Tarjan and Anthony E Trojanowski. Finding a maximum independent set. *SIAM Journal on Computing*, 6(3):537–546, 1977. doi:10.1137/0206038.

40 Marko Vukolić et al. The origin of quorum systems. *Bulletin of EATCS*, 2(101), 2013. URL: http://eatcs.org/beatcs/index.php/beatcs/article/view/183.

41 Gavin Wood. Ethereum: A secure decentralized generalized transaction ledger. White paper, 2015.

42 Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan-Gueta, and Ittai Abraham. Hotstuff: BFT consensus with linearity and responsiveness. In *PODC*, pages 347–356. ACM, 2019. doi:10.1145/3293611.3331591.