# Enhancing Accountability and Trust in Distributed Ledgers[*]

Maurice Herlihy
Brown University and Oracle Labs
mph@cs.brown.edu

Mark Moir
Oracle Labs
mark.moir@oracle.com

June 29, 2016

**Abstract**

*Permisionless* decentralized ledgers ("blockchains") allow anonymous participants and avoid control or "censorship" by any single entity. In contrast, *permissioned* decentralized ledgers allow only explicitly authorized parties to maintain the ledger. Permissioned ledgers support more flexible governance and a wider choice of consensus mechanisms.

Both kinds of decentralized ledgers may be susceptible to manipulation by participants who favor some transactions over others. The real-world accountability underlying permissioned ledgers provides an opportunity to impose fairness constraints that can be enforced by penalizing violators after-the-fact. To date, however, this opportunity has not been fully exploited, unnecessarily leaving participants latitude to manipulate outcomes undetectably.

This paper draws attention to this issue, proposes design principles to make such manipulation more difficult, and provides an overview of specific mechanisms to make it easier to detect when violations occur. More details are presented and related work is discussed in [5].

## 1 Introduction

In *permissionless* implementations, such as Bitcoin [7], anybody can participate in maintaining the ledger, and anyone can generate addresses and keys for send and receiving bitcoins. By contrast, in *permissioned* implementations, participation is controlled by an authority, perhaps one organization, perhaps a consortium.

Permissionless ledgers seek to ensure that nobody can control who can participate, a property often called *censorship resistance*. By contrast, permissioned implementations explicitly permit some forms of censorship: for example, permitting compliance with "know your customer" regulations that exclude known money-launderers from financial markets. Even so, many distributed ledgers—including permissioned ones—would benefit from *fairness* guarantees. For example, one client's proposed transactions should not be systematically ignored or delayed, or one client's transactions should not be systematically scheduled just after another's competing transaction (say, front-running a stock purchase).

In permissionless ledgers, such as Bitcoin, an inherent lack of accountability makes fairness policies difficult to enforce. For example, in a *mining cartel attack* [2, 4, 8], a group of Bitcoin miners ignores transactions proposed by miners which are not members of that group. It can be difficult to detect such behavior, and even if detected, the cartel members suffer no effective penalty.

*In principle*, permissioned ledgers make it easier to define fairness policies and to hold nodes accountable for violations: once exposed, a violator may lose a deposit, it may be expelled from the ledger, or it may be sued. In practice, however, reducing the opportunities for fairness violations, and detecting them when they occur, is a non-trivial problem that requires systematic scrutiny.

[*]Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Our contribution is to call attention to this issue, and to propose principles for designing permissioned decentralized ledgers to hold participants accountable for fairness policy violations. Our approach is consistent with the vision of Yumerfendi and Chase [12], who propose that accountability be considered a "first-class design principle" for dependable network systems.

In [5], we illustrate our ideas by describing a number of modifications to the open-source Tendermint [11] ledger system. However, we believe our principles are applicable to a wide range of ledger implementations. We hope that both the general principles and the specific techniques we introduce will serve to highlight the importance of fairness properties in distributed ledgers, and contribute to effective mechanisms to enforce them.

Our approach is based on the following design principles.

- Each non-deterministic choice presents an opportunity to manipulate outcomes. As much as possible, non-deterministic choices should be replaced with deterministic rules, and mechanisms for detecting violations and holding the culprits accountable should be available.
- Non-deterministic choices that cannot be avoided should be disclosed in an auditable way, ensuring that they cannot be altered later, and that unusual statistical patterns can be detected.
- As much as possible, mechanisms that enhance accountability should be kept off the system's critical path in order to avoid imposing substantial costs on normal-case execution by honest participants.

Sometimes, participants can be held accountable right away: posting *proof* that a participant cheated may directly cause that participant to be penalized or expelled. Often, however, participants can be held accountable only after-the-fact. For example, sometimes one can detect that one of two interacting parties cheated, but it may not be immediately clear which one. Sometimes individual violations may be indistinguishable from legitimate non-determinism. In both cases, reducing opportunities for non-deterministic choices and systematically logging remaining ones can reveal suspicious long-term trends.

Section 2 provides background on Tendermint. Section 3 overviews the mechanisms we have suggested to enhance accountability and trust. In [5], we present these in more detail, and discuss related work and issues including performance, potential manipulations by nodes when accepting transactions and proposing new blocks, resolving differences and integrating external decisions, and fault tolerance.

## 2 Tendermint

We overview relevant aspects of the Tendermint [11] version we considered; for more details, see [9].

Tendermint employs a peer-to-peer network in which nodes gossip: *transactions* submitted by clients, *blocks* of transactions proposed by *proposers*, and various messages used by the consensus protocol used to reach agreement among nodes on successive *next* blocks in the *chain* (blockchain). Tendermint's consensus mechanism is a variant of the PBFT Byzantine agreement algorithm [1, 10]. To add a new block, a *proposer* proposes a block, and *validator* nodes vote whether to accept it.

Honest validators vote only for blocks that satisfy certain validity conditions: for example, each block must include a cryptographic hash of the previous block, making it essentially impossible to change a block already committed into the blockchain. Furthermore, each transaction's preconditions are validated, such as sufficient balances to cover tokens spent by the transaction, valid signatures for each account from which it spends, and the "nonce" value for each such account is in correct sequence following previous transactions from that account. Nonces prevent previously executed transactions being reused by "replay attacks".

Transactions can transfer tokens between accounts, and can create and invoke *smart contracts*, expressed as Ethereum [3] virtual machine byte code; smart contracts can store value and state and provide rules for how these are updated and transferred by transactions.

A transaction enters the network via an RPC call to a node; we call it an *acceptor* node. Each node has a local *mempool* that stores transactions received but not yet seen included in a committed block. An acceptor node validates each transaction as described above before adding it to its mempool. Similarly, a node receiving a transaction from a peer validates it and appends it to its mempool. Each node has a “peer broadcast” routine for each of its peers, which periodically (every 1ms by default) takes a snapshot of the mempool, and gossips transactions from this snapshot to the peer in the order they appear; preserving this order ensures that nonce values remain in correct sequence.

When a node receives a newly committed block, it removes from its mempool all transactions included in the block, as well as transactions invalidated by the transactions included in the new block.

To construct a new block, a proposer takes a prefix of its mempool transactions, in mempool order, thereby preserving correct nonce order of transactions spending from the same account.

Here some opportunities to violate fairness.

**Censorship** A proposer could pretend it never received a transaction (directly or indirectly).

**Selective inclusion** A proposer could delay a transaction by exclusing it from the proposed block.

**Order manipulation** A correct Tendermint proposer proposes transactions within a block in the local mempool order. Nevertheless, a dishonest proposer could reorder transactions without fear of detection.

**Transaction Injection** After receiving a transaction from a peer, a proposer (or its ally) could create another transaction and order it first. For example, a node that has been bribed by an unscrupulous hedge fund might observe a buy order by a pension fund, and inject a buy order of its own before it, immediately reselling that stock to the pension fund at a markup, a practice known as *front-running*.

The following sections provide an overview of specific changes to Tendermint to protect against these problems; they are described in more detail in [5].

## 3 Design Overview

Ideally, any attempt to manipulate transaction outcomes as described above would result in undeniable proof of misbehavior. Failing that, evidence that can be accumulated and analyzed after-the-fact is desirable.

We can make fairness violations much more difficult to achieve without detection by imposing deterministic rules governing the order in which transactions are propagated (making “missing” transactions apparent), which transactions are included in each proposed block, and the order in which they appear. These forms of accountability are achieved by requiring nodes to regularly disclose auditable state information.

**Transaction Ingress** When a Tendermint node accepts a transaction, it returns a *receipt* to the caller. We augment this receipt to include *acceptor data*, which includes an *acceptor ID*, a *sequence number*, and an *acceptor signature* that covers both the transaction and the acceptor ID and sequence number, thus providing undeniable proof that the transaction has been accepted into the network; together with additional protocol extensions described below, it also enables detection of various attempts to manipulate transaction outcomes.

**Accountable Legitimate Censorship** When a node has a legitimate reason for censoring a transaction (for example, it spends from an account on a government blacklist), the node cannot simply drop the transaction. Instead, it appends an explicit, signed indication that the transaction is invalidated, perhaps including a justification. The explicitly invalidated transaction is forwarded as usual, and is eventually put in a block and committed the the chain, making the censorship procedure transparent and accountable.

**Transaction Propagation** In contrast to Tendermint, transactions are validated only upon acceptance and when a proposer includes it in a block (either processed or explicitly rejected). Thus, nodes have no excuse to drop transactions until they have been explicitly processed.

To minimize opportunities for nodes to drop or reorder transactions, we impose the following rule:

**Transaction propagation rule** When a node accepts a transaction from a client or receives one from a peer, it must send that transaction to each of its peers, unless the transaction has already been included in a committed block. All such transactions must be sent in the same order to each peer, in the same order they were received from each peer, and preserving the acceptor ID order for transactions from the same acceptor. There is one exception: a transaction received from multiple peers should be included in the outgoing order once only, and later duplicates are dropped.

We further require proposers to choose transactions for inclusion in a block in the order received.

Tendermint's *implementation* follows these rules: transactions received are appended to the mempool, transactions are gossiped in order from the mempool, and a proposer chooses a prefix of its mempool to include in a new block. However, Tendermint does not limit nodes' ability to deviate from the rules without detection, nor to hold nodes accountable if they do. These are the goals of the extensions we describe.

We describe the proposed mechanisms only briefly here; additional details can be found in [5].

**Data structures** We replace Tendermint's mempool with two data structures: a node's *accepted transaction queue* holds the sequence of transactions that it has accepted but has not yet seen included in a block, and its *outgoing queue* orders transactions for gossip and for inclusion in proposals.

A node's outgoing queue is represented as a key-value store implemented using a dynamic Merkle tree [6]. The keys are integers that order transactions in the outgoing queue, and the values are transactions. This allows nodes to provide concise summaries (Merkle roots) of "snapshots" of their outgoing queues at various times, and to prove the accuracy of subsequent responses to requests for more detailed information about the content of the outgoing queue at those times.

**Communication** A node gossips transactions to its peers using a One-Way Accountable Channel (OWAC) protocol, which is presented in detail in [5]. Briefly, the OWAC protocol allows a receiver to check that the sender follows rules (such as preserving acceptance order) when gossiping transactions, and to provide the sender with cryptographic hashes proving confirmation of transactions, together with Merkle roots representing the receiver's outgoing queue at various times. These mechanisms support accountability in various ways. For example, if the receiver were to reorder transactions it received from a peer, the receiver could prove that it did so, citing the receiver's signed confirmation of the order in which transactions were sent and the contents of the receiver's outgoing queue. OWAC messages can always be piggybacked on messages that are sent anyway in the underlying system (though it does increase message size somewhat).

**Deterministic choices** If a node reports inconsistent outgoing queue summaries to different peers, this could be proved later. Thus, nodes have an incentive to follow the rules and report consistently to all peers. Even so, the OWAC mechanism used for communication *between* peers cannot prevent a node from manipulating the order in which it receives transactions from *different* peers.

We therefore impose additional rules to further limit a node's ability to manipulate transaction outcomes; compliance with these rules can be checked by peers, enabled by the outgoing queue snapshots communicated via the OWAC protocol. First, we require a proposer to select a *prefix* of its outgoing queue to form a block proposal; the length of this prefix is determined by policy, not by non-deterministic choice. Second, we impose deterministic rules on the *order* in which these transactions are included in the block, effectively eliminating a proposer's ability to favour one transaction over another. The transaction order is beyond anyone's control. This can be achieved in a variety of ways, such as using a pseudo-random number generator seeded by a combination of previous block hashes and transaction signatures. Simply ordering transactions randomly, however, may violate the constraint that transactions spending from the same account must be processed in nonce order. We propose a straightforward deterministic solution to this problem in [5].

## 4 Concluding remarks

Designing distributed ledgers to be resistant to internal fairness violations is a complex and important problem, and this paper is only a first step. We exploit the common-sense observation that permissioned ledgers, which have recourse to real-world measures such as fines, expulsions, or legal action, can rely on unobtrusive, after-the-fact auditing and statistical analysis to detect and deter misbehavior, especially repeated misbehavior. By reducing the number of non-deterministic choices, auditors have fewer possibilities to consider when anomalies are detected. To this end, we have proposed several deterministic mechanisms intended to make violations more cumbersome and easier to detect. There are many directions for future work. Can we devise more rigorous, more lightweight mechanisms? What other kinds of fairness violations need to be deterred? Are there theoretical bounds on the kinds of fairness properties that can be monitored, and the cost of such monitoring?

## References

[1] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.

[2] N. T. Courtois and L. Bahack. On subversive miner strategies and block withholding attack in bitcoin digital currency. *CoRR*, abs/1402.1718, 2014. `http://arxiv.org/abs/1402.1718`.

[3] Ethereum. `https://github.com/ethereum/`.

[4] I. Eyal and E. G. Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *Financial Cryptography and Data Security - 18th International Conference, FC 2014, Christ Church, Barbados, March 3-7, 2014, Revised Selected Papers*, pages 436–454, 2014.

[5] M. Herlihy and M. Moir. Enhancing accountability and trust in distributed ledgers. `http://arxiv.org/abs/1402.1718`, June 2016.

[6] R. C. Merkle. A digital signature based on a conventional encryption function. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, CRYPTO '87, pages 369–378, London, UK, UK, 1988. Springer-Verlag.

[7] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. May 2009.

[8] RHorning. Mining cartel attack. `https://bitcointalk.org/index.php?topic=2227.0`, Dec. 2010.

[9] Tendermint. `http:/https://github.com/tendermint/tendermint/wiki`.

[10] Tendermint. `http://tendermint.com/posts/tendermint-vs-pbft/`. Retrieved 6 February 2016.

[11] Tendermint. `http:/https://github.com/tendermint/tendermint/wiki`, Oct 2015. Commit `c318a227`.

[12] A. R. Yumerefendi and J. S. Chase. The role of accountability in dependable distributed systems. In *Proceedings of the First Conference on Hot Topics in System Dependability*, HotDep'05, pages 3–3, Berkeley, CA, USA, 2005. USENIX Association.