# Modeling and Verification of Keeta's Two-Phase Consensus Protocol

xescure

xescure@xescu.re

2026

Cross-border financial settlement remains slow and costly, with structural inefficiencies rooted in the correspondent banking system. Keeta is a recently released blockchain network designed for high-volume global payments, employing a novel two-phase client-directed consensus algorithm. Despite its promise, the algorithm has only been described informally in a whitepaper and has not been subject to formal analysis. This paper presents the first formal specification of Keeta's two-phase consensus algorithm, written in Quint and model-checked with TLC under a Byzantine fault model. It establishes that the algorithm preserves agreement under a fixed, equally weighted set of representatives, argues that this safety guarantee generalizes to any number of representatives, and exhibits conditions under which a contended account can be driven into a permanent lockout. Whether agreement survives Keeta's live, stake-weighted voting is left open. The resulting model is machine-checkable and serves as a reusable artifact for future research.

**Keywords:** Keeta, consensus algorithms, formal verification, Quint, TLA$^+$, Byzantine fault tolerance, blockchain, DAG, distributed ledger technology

# Contents

# 1 Introduction

Even today, cross-border financial settlement remains slow and costly relative to domestic payments. Remittance costs average around 6% of the transfer amount [9, Table 12, p. 29], while only a third of retail cross-border payments settle within an hour [9, Table 8, p. 23]. These are structural inefficiencies attributed to correspondent banking, the dominant backend for cross-border settlement, where banks settle transactions through pre-funded accounts they hold with each other. Since each pair of banks must establish these accounts bilaterally, the resulting network is fragmented and capital inefficient. Not only does correspondent banking tie up working capital in pre-funded accounts, but it also introduces intermediaries for transactions between parties without a direct relationship.

Distributed ledger (DLT) systems have been proposed as a structural solution [3], enabling settlement on a shared ledger that eliminates pre-funded bilateral accounts and allows transactions between potentially distrusting parties without a pre-existing relationship. Despite this promise, real-world adoption in financial infrastructure remains minimal, partly due to scalability limitations, a lack of compliance controls, and a fragmented landscape of competing standards, which prevents the network effects that traditional payment systems benefit from.

Keeta is a recently released compliant blockchain network designed to facilitate global financial transactions at scale [10]. Its highly parallelizable architecture makes it a good fit for high-volume global payments. Keeta's ledger stores a chain of blocks (a blockchain) for each account in its system, pushing it closer to a directed acyclic graph type DLT than a traditional blockchain system.

Keeta employs a novel two-phase client-directed consensus algorithm for confirming transactions on the network. To be considered final, a block undergoes two phases: first, the client presents it to each validator node, called a *representative*, to collect temporary votes; upon collecting enough temporary votes, the client requests permanent votes from the same representatives. With enough permanent votes, a transaction is considered final and is submitted to the peer-to-peer network by the client [10]. This ensures only valid operations are ever added to the global state, even in the face of malicious representatives, up to a threshold.

Consensus algorithms underpin the safety and reliability of blockchain systems — a flaw can result in double spends or permanently stalled transactions. Unlike cryptographic primitives such as signature schemes, whose security

properties are easily reusable, a consensus algorithm's correctness is specific to its own design. Whether transactions are safe from double spends, and whether they will ever reach finality, depends on the exact rules of how nodes vote, what constitutes an agreement, and how the protocol behaves when some participants act maliciously. Thus, these properties must be established independently for each new protocol. Keeta's protocol has been described informally in the whitepaper [10] but has not been subject to formal analysis. While informal descriptions are useful for introducing a protocol, they are not required to be precise about edge cases and failure modes. Without such an analysis, the correctness of the protocol cannot be established beyond reasonable doubt, and any deployment of Keeta in real financial infrastructure is based on unverified assumptions.

This paper addresses that gap with a formal, machine-checkable specification of Keeta's two-phase consensus algorithm. Where the whitepaper states the protocol-level invariants only informally, the model makes them explicit, and serves as a reusable artifact for future research.[1] It is used to test the algorithm's safety under Byzantine faults, and to probe whether a contended account can be driven into a permanent lockout.

In light of the above, the work is guided by three research questions:

1. What does a machine-checkable formal specification of Keeta's two-phase consensus look like?
2. Does the specified protocol preserve safety (agreement) under up to $f$ Byzantine representatives, and does that guarantee extend beyond the checked configurations?
3. Can the protocol reach a permanent lockout on a contended account, and under what conditions?

The Keeta whitepaper is the primary source for the formalization [10]. The client's source is partially public as a compiled bundle, from which most of the protocol logic can be reverse-engineered, though some components such as the storage drivers are not included. Remaining ambiguities were resolved directly with the protocol's designers.

[1] The Quint model and the verified configurations are available at https://github.com/xescure/keeta-consensus-spec.

## 2 Background

Society entrusts its most essential processes to computers. Some, such as those in military, healthcare, and finance, are too critical to rely on a single server for. Distributed computing allowed such systems to tolerate individual crashes and stay active despite natural disasters, human error, and hardware failure. The vast majority of real-world distributed systems have been crash-tolerant systems. Byzantine fault-tolerant services, ones resistant to malicious actors, have been rarely used due to their communication overhead and complexity, among nontechnical reasons like industry inertia. Yet the environments that demand Byzantine fault tolerance are ever so prevalent. Global finance is a clear example. Institutions that settle with each other do not fully trust each other, so no single party can be trusted to maintain the ledger.

If no single party can be trusted with the ledger, then every participant must hold a copy, and the copies must agree. This is the consensus problem, a foundational problem in distributed computing: A set of nodes each propose a value, and all of them must settle on one of them. Deciding which entry is added into a financial ledger is an instance of this problem.

A protocol achieves consensus if it guarantees three properties. *Agreement*: no two correct nodes decide on different values. *Validity*: the final value was proposed by a node, which rules out the trivial protocol that agrees on a pre-defined value. Both are safety properties. *Termination*: every correct node eventually decides. This is a liveness property, since a protocol that never decides is still safe. A double spend is a safety violation; a transaction that never finalizes is a liveness failure.

The Byzantine Generals Problem [13] was given its first practical solution by the Practical Byzantine Fault Tolerance (PBFT) algorithm [6]. It brought Byzantine tolerance to asynchronous systems like the internet. PBFT replicates a series of events across $3f + 1$ nodes, where f nodes could behave arbitrarily or maliciously without impacting the system's functioning. Decisions require a quorum of $2f + 1$ votes. Any two quorums of this size share at least one honest node, so no two conflicting decisions can both reach a quorum. A designated leader puts every operation into a single global order, which the nodes confirm over three rounds of voting. Descendants of PBFT, such as Tendermint and HotStuff, became the consensus core of blockchain systems and inherited this design where every transaction is ordered by a leader and agreed upon in multiple voting rounds.

However, ordering every operation in relation to every other is an expensive process. A total order serializes all transactions into a sequence. Parallel

processing can be achieved, but only where transactions happen to not conflict. Payments rarely do, and the order between payments of unrelated accounts is irrelevant to the resulting state. FastPay, a client-driven high-throughput payment system designed at Facebook, showed that the only ordering a payment requires is on the sender's own account [2]. In FastPay, clients contact each validator node, called authorities, and each responds by signing the transfer order. A quorum of 2f + 1 signatures forms a transfer certificate, which proves finality. The client must still take the certificate to each authority for propagation, and consequently to unlock voting for its future transactions. Since accounts are independent, authorities can process them in parallel, enabling nodes to scale horizontally. There is no global sequence that a leader would need to propose, and the failure of any authority leaves the network completely unaffected. Moreover, FastPay's guarantees hold in a fully asynchronous network. Neither its safety nor its liveness require any assumptions about message delays or clocks. However, the network's robustness comes at a cost. If a malicious or faulty client sends conflicting transactions to different authorities, then its account may permanently be disabled [2]. Additionally, FastPay's design supports only simple transfers of a single asset. Operations that span multiple accounts, such as atomic swaps, are not supported. A network that is restricted to a single asset, and offers no protection against counterparty risk, cannot serve as general financial infrastructure, and is simply a supportive settlement layer.

## 2.1 Formal Verification and Model Checking

Blockchain networks must stay safe under every possible case — no matter which messages are delayed, or which expected failures take place. Simple testing can only cover so many cases, so a more robust approach is needed to reason over every possible behavior. Deductive proofs can establish a system's correctness, but they take considerable effort and must be redone when the protocol changes. For an evolving system like Keeta, a more flexible approach is required. Model checking is an automatic verification technique for software systems that exhaustively explores possible states and determines if certain properties hold. When they do not, the results are accompanied by concrete counterexamples to aid understanding. Model checking has been applied to comparable Byzantine fault-tolerant protocols [11, 15], including Tendermint [5, 12].

The model is specified in Quint, a specification language in the TLA$^+$ family. A consensus protocol can naturally be expressed as a state machine, where a state captures the system's components — blocks, votes, and account ledgers

— and the transitions describe how those components change as the protocol runs. This is exactly what languages in the TLA$^+$ family are designed for. Quint provides the same logic as TLA$^+$, but with a programmer-friendly syntax, static typing, and an integrated REPL, while staying compatible with established tooling, such as the TLC and Apalache model checkers. Because Keeta is still evolving, a model that engineers can own and edit is more valuable than a one-off artifact that goes out of date as the protocol changes. Such a model is the reusable artifact this work contributes.

## 3 Keeta Protocol

Keeta is built for global financial transactions at scale, claiming millions of transactions per second with sub-second deterministic finality. Instead of building one global sequence of events as a global chain of blocks, Keeta separates transactions for each account. Two of the key characteristics that distinguish blockchain networks from one another are the data structure used to store the ledger and the consensus algorithm used to agree on its state.

In terms of data structure Keeta has taken inspiration from the Nano cryptocurrency and given every account an independent blockchain. Each block extends its account's chain by referencing the hash of its predecessor, giving every account a locally ordered history. The ledger is the collection of these account chains. Some operations, such as atomic swaps, touch more than one account at once, linking otherwise independent chains into a directed acyclic graph. Ordering also arises from dependency, as when a payment that draws on incoming funds becomes valid once those funds are confirmed. Each transaction is limited by the transactions it depends on, and the resulting order emerges from the dependencies.

Separating accounts breaks up the single sequence other blockchains must agree on, and several properties follow from this architecture. With no global order there is no leader to rely on for coordination, which reduces the ability of a slow or faulty leader to delay progress, improving worst-case behavior [1]. Unrelated blocks do not sit in a shared mempool where they are vulnerable to Maximal Extractable Value (MEV) attacks, such as frontrunning, where an attacker observes a pending transaction and submits their own first to make a profit [8]. Without a global sequence of events, transaction dependencies are more straightforward to decouple, resulting in more parallelizable software, horizontal scaling, and a higher transaction throughput. As computational power can be traded for capacity, transaction throughput becomes a commodity rather than a scarce resource [7], and the congestion pricing of conventional

chains is avoided. Operations that span accounts, such as atomic swaps, remain possible, and are described in the next section.

The second distinguishing characteristic is the consensus algorithm. With no global sequence to agree on, Keeta's consensus decides only which blocks become final extensions of their accounts. Understanding the consensus process requires three concepts: operations, blocks, and staples.

An operation [10, Listing 1] is a single action belonging to one account, such as a SEND that withdraws from a balance. Operations are grouped into blocks, one block per account, signed by a key authorized to perform them. A SEND is constrained only by the account it originates from, meaning it requires ordering on that account's chain, but nothing from the recipient. The recipient's balance rises as a result, but no block is written on their chain and no coordination with them is needed. Ordering is up to the sender alone.

However, some operations cannot be settled from one side. A staple is a set of blocks bundled together for voting, that either succeeds or fails as one. A client may combine any valid signed blocks into a staple, which is what lets a swap settle atomically. The same mechanism extends further, where any application that must update several accounts together, such as a smart contract, can rely on cross-account staples.

A staple becomes final through a vote. This process is driven by the client, which gathers votes from the representatives, appends them into the staples, and publishes the finalized blocks to the network. Representatives are the voting nodes with the authority to decide on the network's state. Depending on the network's configuration, a representative's vote is weighted according to the stake delegated to it [14], meaning agreement is measured by weight rather than by a count of nodes. A quorum is reached once votes representing more than two-thirds of the total weight support a staple.[2]

The vote consists of two phases: temporary and permanent (Figure 1). First, the client sends the staple to the representatives, each of which validates it and returns a temporary vote. Validation consists of ensuring the operations in each block are valid, e.g. SENDs are covered by sufficient balances, and that each block is signed by an actor authorized to submit them. Once temporary votes exceeding the quorum have been collected, they are added to the staple, and the client presents the staple to the same representatives to, in turn, receive permanent votes. A quorum of permanent votes makes the staple final, and

[2]The whitepaper only describes this quorum as weighted and adaptive and does not give a fraction. The two-thirds threshold is inferred from the reference client: https://github.com/KeetaNetwork/keetanet-client/blob/81a87826519f61fa4ae4f9930f50ae5772e624a7/client/index.js#L69588-L69594.

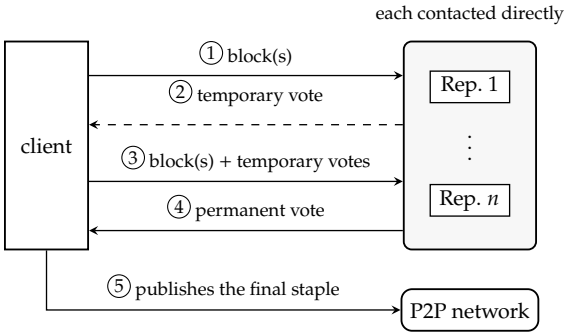


**Figure 1:** Keeta's two voting phases, driven by the client. Dashed replies are temporary votes.

the client publishes it to the P2P network, where it is stored and relayed by the network's nodes.

The temporary votes collectively show that the network is ready to finalize a set of blocks, and this readiness is self-evident from the votes themselves. Any representative can verify the temporary votes from their signature, timestamp, and their own internal clock, without contacting other representatives. Therefore, the number of messages necessary to finalize a staple scales linearly with the count of representatives. In contrast to temporary ones, permanent votes are final, irrevocable, and once they reach the quorum threshold, the staple is finalized and its blocks are permanently added to their accounts' chains.

Compared to FastPay, this design makes client failure and contention much less likely to permanently block an account. If a client fails to gather enough temporary votes for any reason, it can simply retry after the votes expire. If the failure is during the second phase, the client can combine the successful permanent votes with fresh temporary ones, to reach permanent finality and move the account's state forward.

This recovery is what enables contention at all, and contention is inherent when multiple parties can submit blocks on the same account. With pull payments, for instance, an account authorizes another party to take payments from it, and both could attempt to submit a block at the same time. Other features, such as atomic swaps and delegated custody, serve common financial needs like delivery-versus-payment settlement, while also introducing the same shared access to accounts.

Together, these pieces point to a core design principle that underpins Keeta: Optimistic Concurrency Control, or OCC. Keeta applies OCC principles at

the account level, so conflicts (and the retries inherent to this architecture) arise only where multiple parties can write to the same account. Transactions proceed normally without locking the affected accounts, and only the rare conflicts need to be retried. The wager is that this fits the properties of finance. Most payments are independent, so conflicts are infrequent, and the system can run almost entirely parallel. Even when they happen, conflicts are localized, and do not affect the rest of the system. Keeta's designers made the judgment that this architecture would not only cover the needs of blockchain-based finance, but do so reliably enough and at global scale. This choice to operate without ordered consensus sets Keeta apart even within its own lineage. Sui shares the same foundations for its owned objects [4], but still relies on full consensus for shared state. Whether this bet holds at scale is not yet established in wider use. This work aims to contribute to that question by examining the protocol's guarantees formally.

## 4 Formal Model

Having described the protocol, this section formalizes its consensus algorithm as a state machine. Concepts orthogonal to consensus are deliberately scoped out, including block content validation — balances and permissions — that sits on a different layer. Cryptographic signatures remain in the model, treated as unforgeable (Table 1). The model checks agreement, meaning no two correct representatives decide on different values, and whether a contended account can be locked out of progress.

The model's state captures representatives and their versions of the ledger. Following Keeta's own terminology, the *main ledger* contains finalized blocks, and the *side ledger* keeps track of the representative's own issued votes. Blocks contain no payloads; they simply carry a unique identifier and link to their parent. The system models a single account, which begins with an origin block and forms a linear chain with each subsequent one. Valid cryptographically signed blocks and votes are tracked in global oracles (`signedBlocks`, `signedVotes`). These are used solely for the purposes of simulating signatures and to be used in invariants; they are not allowed to expose further information to representatives.

The state transitions are the actions representatives take, normally in response to a client's request. Representatives will issue temporary votes (`issueTemp`) and, when presented with a sufficient number of temporary votes, they will provide permanent votes (`issuePerm`). Moreover, with enough permanent votes a staple can be applied to each representative's main ledger

(`applyStaple`). Once a staple is finalized, the lifecycle ends. In real Keeta, representatives then publish the blocks and votes to the P2P network. The model omits this step, as propagation concerns availability and liveness, not safety. The actions are exposed directly to the model and are not driven by modeled clients. Because of this, the checker explores not only what a single client would do, but what any number of clients — parallel, competing, or malfunctioning — could attempt.

The model abstracts several aspects of the protocol to keep the state space finite and focused on consensus. Table 1 lists them against their real counterparts. The safety results depend on two of these abstractions. Signatures are treated as unforgeable, and the representative set is fixed and equally weighted, which is the fixed quorum denominator the intersection argument assumes. In the live protocol the weights are not fixed, and what that means for safety is discussed in Section 6. Expiry is the hardest to capture, and the model simplifies it twice. First, as a verification optimization, temporary votes may lapse in any order rather than in the order they were issued. This covers a superset of the real states, so results that hold carry over, but a violation may come from an impossible order and has to be cross-checked. Second, a vote lapses for every representative at the same moment, with no clock drift between them. This cannot break safety, but it can hide the lockouts that only clock skew would produce.

**Table 1:** Aspects of the protocol abstracted in the model, with their consequences.

| Aspect | Real Keeta | In the model | Consequence |
|---|---|---|---|
| **Signatures** | currently ed25519 / ECDSA certificates | oracle for signed votes; assumes no forged votes | cryptography assumed unbreakable |
| **Voting weight** | dynamic, based on token balances (`SET_REP`) | fixed set, equal weights | sound for the fixed weight set; dynamic weights are concerning in the live system |
| **Vote types** | permanent and temporary types derived from validity time | explicit vote kind, `Temp` or `Perm` | no apparent cost |
| **Temporary vote expiry** | five-minute lifetime | `Temp` votes can lapse in any order | covers a superset of the real state space, but improves TLC performance considerably; violations to be cross-checked |
| **Clock synchronization** | reasonably synchronized clocks across honest representatives | synchronized expiry of temporary votes | not applicable to safety; may hide skew-induced lockouts |

| Aspect | Real Keeta | In the model | Consequence |
|---|---|---|---|
| **Ledger** | DAG consisting of account blockchains | one account chain only | cross-account operations not modeled; the single-account contention the results depend on is preserved |
| **Block production** | unbounded | capped by `MaxBlocks` and `MaxForkWidth` | finite instances; generalized separately |
| **Block content** | balances, permissions, operations | not modeled | out of scope for consensus |
| **P2P network** | P2P delivers staples between representatives | P2P not modeled; staples are delivered directly to each representative | unsound for liveness; otherwise covers the same state space |
| **Block expiry and rescue** | blocks expire, with a dedicated rescue path | not modeled | likely introduces lockout possibilities |

The model checks agreement and lockout-freedom, together with a structural validity check, each expressed as an invariant over the state:

- **Agreement** is captured by the invariant `NoFork`. It rests on the protocol's local rule that a compliant representative never issues two competing votes [10], an act called *equivocation*. Together with the intersection of any two quorums, this rules out a fork across the network. The full argument is developed in the verification section.
- **Validity** holds structurally. Nothing is finalized that was not legitimately produced. Its role is to rule out a degenerate model that would satisfy agreement trivially, such as one that finalizes nothing at all. Keeta's fuller notion of validity, in which a client authored and signed a block and its operations respect balances and permissions, is out of scope.
- **Lockout**: the model checks whether a permanent lockout is reachable under client contention, when two parties compete to write to the same account. This is the scenario FastPay is vulnerable to.

Finally, the model includes some well-formedness invariants that give the model redundant internal cross-checks. These cover type-correctness, that the state contains no block or vote that was not actually produced, and that no honest representative ever equivocates (`PerRepForkFree`).

## 5 Verification

The specification was checked with two model checkers. TLC [16] is an explicit-state checker that enumerates the reachable states one by one, so on a bounded configuration it visits every reachable state and the check is exhaustive. Apalache is symbolic; it compiles the specification to a logical formula that an SMT solver (Z3) tests for violations within a bounded number of steps.

TLC is the model's primary checker. The model was optimized for its explicit search, and every result in this section comes from TLC. Apalache struggled with the set-heavy architecture of the specification, but its inductive verification may still prove useful for future research (Section 5.3).

The model has four parameters, fixed by each configuration:

- **Representatives** (`Reps`) are the voting nodes. The quorum threshold is more than two-thirds of their number $n$.
- **Byzantine representatives** (`Faulty`) are the ones that may equivocate. The remaining $n - f$ are honest.

**Table 2:** Paper notation and the corresponding definitions in the model (`Keeta.qnt`).

| Paper | Model | Paper | Model |
|---|---|---|---|
| $R$ | `Reps` | voters($c$) | `permsFor` |
| $H$ | `Honest` | Final($c$) | `isCommitted` |
| $F$ | `Faulty` | NoFork | `NoFork` |
| Q($S$) | `isQuorum` | $h_r$ | `repHead` |
| $V$ | `signedVotes` | $K_r$ | children of `repHead` |
| perm($q, c$) | a `Perm` vote | Free($r$) | `freePool` |
| parent($\cdot$) | block `parent` | B($r, c$) | `permOrCouldPerm` |
| $o$ | `OriginId` | CanProgress | `CanProgress` |

- **Block bound** (`MaxBlocks`) caps how many blocks are proposed beyond the origin, which restricts the length of the chain.
- **Fork width** (`MaxForkWidth`) caps how many competing children a block may have, which sets how much contention the checker explores. Values at or above `MaxBlocks` make no difference, since the block limit is reached first.

Most configurations in Table 3 use $n = 4$, with the largest reaching $n = 7$; Section 5.3 discusses how far exhaustive checking scales.

The verification uses the notation below, mapped to the model in Table 2. Write $R$ for the representatives, $F \subseteq R$ for the Byzantine subset, and $H = R \setminus F$ for the honest ones, with $n = |R|$ and $f = |F|$; the network satisfies the fault bound $n \geq 3f + 1$, which Section 5.1.1 derives from the quorum rule. A set $S \subseteq R$ is a *quorum* when

$$\mathrm{Q}(S) \;:\equiv\; |S| > \tfrac{2}{3}\, n.$$

Let $V$ be the set of signed votes, with $\mathrm{perm}(q, c) :\equiv (q, c, \mathsf{P}) \in V$ when representative $q$ holds a permanent vote for block $c$. Blocks form a tree under parent($\cdot$) rooted at the origin $o$.

## 5.1 Safety

Agreement holds in every configuration checked up to the fault bound. The `Safety` invariant pairs agreement (`NoFork`) with the well-formedness cross-checks, and it was exhausted at $f = 1$ over 10.5 million distinct states with no violation.

**Table 3:** Model-checking results from TLC. *Bounds* caps blocks and fork width. Counterexample rows report only *depth* (—); *trivial* means violated at the initial state; *Progress* is `CanProgress`. †Not guaranteed exhaustive (fingerprint limit).

| $n$ | $f$ | bounds | invariant | states | depth | time (s) | result |
|---|---|---|---|---|---|---|---|
| 4 | 0 | 3/3 | Safety | 1 882 927 | 40 | 46 | holds |
| 4 | 0 | 3/3 | Progress | 1 882 927 | 40 | 38 | holds |
| 4 | 1 | 2/2 | Safety | 198 736 | 27 | 6 | holds |
| 4 | 1 | 3/2 | Safety | 10 469 568 | 39 | 206 | holds |
| 4 | 1 | 4/2 | Safety | 789 207 872† | 51 | 19 786 | holds |
| 4 | 1 | 2/2 | Progress | — | 12 | 3 | lockout |
| 4 | 2 | 2/2 | Safety | — | 13 | 6 | fork |
| 4 | 2 | 2/2 | Progress | — | 0 | 2 | trivial |
| 5 | 1 | 2/2 | Safety | 1 359 876 | 33 | 34 | holds |
| 7 | 2 | 2/2 | Safety | 1 766 433 888† | 45 | 61 358 | holds |

A block is *finalized* once a quorum permanently votes for it:

$$\begin{aligned} \mathrm{voters}(c) &= \{\, q \in R : \mathrm{perm}(q, c) \,\}, \\ \mathrm{Final}(c) &\;:\equiv\; c = o \;\vee\; \mathrm{Q}\big(\mathrm{voters}(c)\big). \end{aligned}$$

The quorum makes no distinction between honest and Byzantine representatives. Agreement (`NoFork`) forbids two distinct finalized blocks under a common parent: for all blocks $b, b'$,

$$\mathrm{Final}(b) \wedge \mathrm{Final}(b') \wedge \mathrm{parent}(b) = \mathrm{parent}(b') \;\Rightarrow\; b = b'.$$

At $f = 2$ the fault bound is exceeded and agreement fails. TLC returns a fork in twelve steps. The two Byzantine representatives each equivocate, permanently voting for both children of a block, so two honest representatives finalize competing children. The two committing quorums intersect only in the Byzantine pair, so no honest representative endorses both blocks, and the local anti-fork rule holds even though the network forks. The trace uses no expiry step, so the fork is caused by equivocation alone and not by the expiry abstraction. This shows the $n = 3f + 1$ bound in concrete terms. With two Byzantine representatives out of four, two quorums can each exceed two-thirds while sharing no honest member.

### 5.1.1 Generalization

The model checks agreement on a handful of fixed network sizes, but there is no reason for the property to depend on that size, and this section argues that it does not. The only thing the protocol establishes is the quorum rule from Section 3, that a staple becomes final once the permanent votes cast for it carry more than two-thirds of the total voting weight. On its own this says nothing about how many representatives may act maliciously. That figure, the fault bound, appears in neither the whitepaper nor the source, and has to be derived from the quorum rule.

The derivation needs two facts. First, an honest representative casts at most one permanent vote among the blocks that extend a given point in an account's chain, so it never votes on two conflicting staples.[3] Second is a counting argument on the quorum. Suppose two conflicting staples both reached finality. Each is backed by permanent votes carrying more than two-thirds of the weight, so together they account for more than four-thirds of a total that is only one whole. Two shares that large must have more than a third of the weight in common, and that shared third is exactly the representatives who voted for both staples. If the malicious representatives carry less than a third of the weight between them, this shared third cannot be theirs alone, so at least one honest representative voted for both, which the first fact rules out. The two staples could not both be final.

The argument holds only while the malicious representatives carry less than a third of the weight. Under equal weights that is fewer than $n/3$ representatives, the familiar $n \geq 3f + 1$, and it is where Keeta's fault bound comes from rather than being assumed. Smaller thresholds tolerate proportionally fewer malicious representatives. Two quorums that each carry more than $t$ of the weight are only guaranteed to share more than $2t - 1$ of it. At a bare majority that guarantee vanishes, and two quorums could meet in a single malicious representative that finalizes both staples. Raising the threshold helps safety but hurts availability, since a quorum must still be reachable while the malicious representatives stay silent, which caps their share at $1 - t$. The two constraints balance at two-thirds, where each breaks at a third of the weight [6]. None of the steps mentioned how many representatives there are, so the same reasoning holds at any network size.

This argument is on paper rather than by the model. `Safety` is checked as an invariant over the bounded configurations of Section 5.1 and is not established

[3] Stated in the whitepaper [10] and enforced in the reference client: https://github.com/KeetaNetwork/keetanet-client/blob/81a87826519f61fa4ae4f9930f50ae5772e624a7/client/index.js#L69119-L69147.

as inductive, so a machine-checked proof for unbounded $n$ is left to future work (Section 7).

### 5.2 Lockouts

RQ3 asks whether a contended account can become permanently stuck. This is checked as a reachability property, `CanProgress`, rather than a temporal liveness claim. In plain terms, an account can still move forward in two ways. Either enough honest representatives are still free to form a new quorum, or one of the competing children can still reach a quorum from the votes already recorded for it and the free representatives. A lockout is when neither can happen. Fix an honest representative $r$ with head $h_r = \mathrm{head}(r)$, and let $K_r$ be its competing children, the blocks $c$ with $\mathrm{parent}(c) = h_r$. Progress is counted conservatively. A Byzantine representative may never issue another vote, so future votes are only counted from honest representatives:

$$\mathrm{Free}(r) = \{\, q \in H : \forall\, c \in K_r,\ \neg\, \mathrm{perm}(q, c) \,\},$$
$$\mathrm{B}(r, c) = \mathrm{Free}(r) \cup \{\, q \in R : \mathrm{perm}(q, c) \,\}.$$

Progress remains possible from $r$ when a fresh honest quorum survives or some child can still reach one:

$$\mathrm{Prog}(r) :\equiv \mathrm{Q}\big(\mathrm{Free}(r)\big) \ \vee\ \big(\exists\, c \in K_r,\ \mathrm{Q}\big(\mathrm{B}(r, c)\big)\big),$$
$$\mathsf{CanProgress} :\equiv \forall\, r \in H,\ \mathrm{Prog}(r).$$

A lockout happens iff neither is possible: some honest $r$ has $\neg\mathrm{Q}(\mathrm{Free}(r))$ and $\neg\mathrm{Q}(\mathrm{B}(r, c))$ for every $c \in K_r$.

Within the checked configurations, with expiry synchronized across representatives and block expiry not modeled (Table 1), `CanProgress` holds at $f = 0$. However, it is violated at $f = 1$ (Table 3), which is still within the fault bound of a network with $n = 4$.

TLC returns the $f = 1$ lockout in eleven steps, and the trace is reproduced as a test in the published model (`lockoutSingleByzantine`). With three honest representatives and one Byzantine, a quorum is three votes:

1. Two clients propose competing children $b$ and $b'$ of the head.
2. Representatives 1 and 2 issue temporary votes on $b$, and representative 3 on $b'$.
3. The Byzantine representative 4 equivocates, issuing a temporary vote for each child.

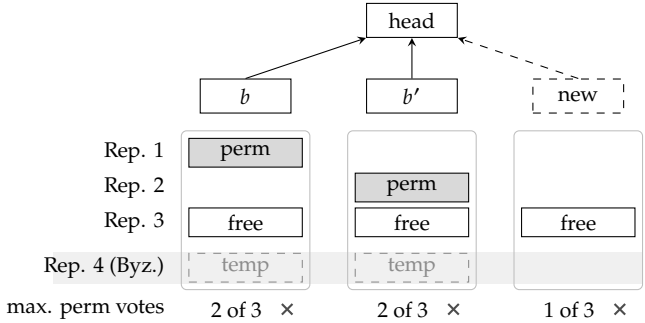


**Figure 2:** The wedged state at $n = 4$, $f = 1$. Each block falls short of the quorum of three permanent votes.

4. On $b$, the votes of 1, 2, and 4 now form a quorum, so representative 1 upgrades to a permanent vote and is pinned to $b$.
5. Representative 2's temporary vote lapses, and it swings to $b'$. The votes of 2, 3, and 4 form a quorum there, and representative 2 upgrades and is pinned to $b'$.

The honest permanent votes are now pinned one against one, with only representative 3 free (Figure 2). Even counting 3, each child can reach two permanent votes of the three required, so `CanProgress` fails.

Every path forward now runs through the Byzantine representative. Representatives 1 and 2 are pinned for good, and 3 can join only one child, so no child can collect three permanent votes from the honest side alone. The Byzantine temporary votes that pinned them cannot close the gap either, since only their issuer could upgrade them, and it may simply never vote again. The account is not deadlocked in the literal sense. A single permanent vote from the Byzantine representative would let a child finalize, but nobody can force it to vote.

Equivocating with permanent votes would not cause a lockout. A signed permanent vote cannot be taken back, which is why $\mathrm{B}(r, c)$ counts them from any issuer, and in the model representative 3 could use the recorded votes to complete a child on its own. Temporary votes leave nothing behind; even the evidence of the equivocation expires with them.

As a final note, the counterexample survives the over-abstraction created by any-order expiry. It contains only a single expiry, so it is not an artifact of the model but a realistic sequence of events.

The problem itself is not unexpected. It is the account contention problem

that Keeta inherits from FastPay, and the whitepaper acknowledges contention on accounts that several parties may write to. The designers' planned mitigation, a checkpoint mechanism, is discussed in Section 6.

### 5.3 Scalability and Tooling

Exhaustive checking is bounded by the state count, which grows steeply with the model's parameters. Each added representative multiplies the reachable states by roughly seven, each unit of fork width by about two, and each unit of depth by over fifty. On a sixteen-core virtual machine (Hetzner CX53), thirty minutes of exhaustive search covers around a hundred million states. The largest safety result, $n = 7$ with $f = 2$, took about seventeen hours over 1.77 billion states, and at that size the search reaches the checker's fingerprint limit and is no longer guaranteed exhaustive. This is enough to exhaust the configurations in Table 3 and to establish safety at the fault bound, but it does not extend to substantially larger sets.

Apalache, the symbolic model checker, was less useful for the same purposes. Its search is bounded by depth rather than state count, and on this model the depth limits are hit early. The jump from depth six to seven took several minutes, and depth eight was never reached within a thirty-minute time budget. The earliest fork in this model appears at around step twelve, well beyond what Apalache managed to reach. This performance bottleneck can presumably be attributed to the set-heavy architecture of the model and its reliance on cardinalities. However, this is just speculation, as Apalache has not been a focus of this research. Apalache is further limited by its inability to scale across multiple cores, while TLC can take full advantage of multithreading.

Apalache's value for future work lies in its inductive invariants.[4] Inductive invariants can establish safety for every representative set at once. Agreement is not inductive on its own, and would need manual strengthening for this to be applicable. This is the established route to a machine-checked proof for unbounded $n$, but given the generalization in Section 5.1.1, it has not been a priority.

## 6 Discussion

Agreement holds in every configuration checked up to the fault bound, and the quorum-intersection argument extends that guarantee to every representative

[4] Quint documents inductive-invariant checking at https://quint.sh/docs/checking-properties#inductive-invariants.

set with $n \geq 3f + 1$, so under the model's assumptions the consensus core is safe and no two conflicting blocks can ever be finalized. Importantly, this is under the assumption of constant, uniform voting weights, the consequences of which are discussed later in this section.

A permanent lockout is reachable, however, so the protocol does not fully escape the availability failure that its two-phase design was meant to address. The conditions for a lockout are narrower, and the likelihood of a well-intentioned, healthy account encountering one is meaningfully lower. This is a real improvement over FastPay, where a single client failing at the wrong moment can permanently disable its own account with no way back [2].

According to the model, a lockout strictly requires a Byzantine representative acting on an already contended account. Empirically, there are other lockout cases hidden away by the model's abstractions, block expiry later in this section and clock skew in Table 1, but these do not meaningfully change the interpretation.

In the live protocol, it is not just temporary votes that expire, but also blocks themselves, which opens another way to reach a lockout. This one needs no Byzantine representative. A misbehaving client could keep creating blocks and gather only a little voting weight for each. A representative will not add a temporary vote to a block unless the votes already on it are worth more than a quarter of the total weight.[5] So a block that never gets past that quarter cannot pick up the new votes it would need once its old ones expire, and it stays stuck. Leave enough blocks stuck this way and the account can no longer move forward. The implementation does have a rescue method that lets representatives vote on expired blocks, but its purpose seems to be making lockouts rarer, not removing them entirely.

The abstraction that matters most is the voting weight, because it is the one that can affect safety. The model gives representatives fixed, equal weight, but Keeta weighs votes by delegated stake in the base token, set through `SET_REP` operations. The base token, like any other token, goes through the same consensus process. Some transactions may be delayed if issues arise with the P2P delivery layer. This is inherent to Keeta's design; order is only enforced when strictly necessary. However, the base token is implicitly required for each transaction. If two representatives see differing versions of the voting weights, it becomes plausible for two staples to each clear the quorum threshold on different representatives. This is a hypothesized mechanism for a safety

[5] `minVotingWeight = totalVotingPower / 4` in the reference client: https://github.com/KeetaNetwork/keetanet-client/blob/81a87826519f61fa4ae4f9930f50ae5772e624a7/client/index.js#L69072-L69079.

(agreement) violation, but one that has not been demonstrated on the model. The model could be extended with dynamic weights to explore whether this issue is already ruled out, or whether other remedies would be needed.

Checkpoints are a planned addition to the Keeta protocol. Their exact implementation is not yet public, but their function can be inferred from Keeta's trade-offs and from the practices of similar protocols, notably Sui [4]. Representatives would periodically agree on a snapshot of the finalized ledger, together with the representative set and voting weights to use until the next checkpoint. This would address both the complications of dynamic representative weights and the recovery of locked-out accounts. Fixing the representative set and its weights for an epoch gives every representative the same quorum denominator, which is close to what this model already assumes, so the reconfiguration risk described above cannot arise within an epoch. Discarding blocks that were proposed but never finalized at each checkpoint clears the wedged blocks behind a lockout, letting a stuck account resume in the next epoch. Sui Lutris uses this epoch structure to bound equivocation lockouts to a single epoch and to reconfigure its representative set. This recovery has a cost. The periodic agreement is itself a step of consensus, which Keeta's per-transaction path deliberately avoids, so checkpoints trade some of that leaderless simplicity for reconfiguration and recovery. The safety question would then move to the checkpoint mechanism itself, which would be the next thing to verify.

## 7 Future Work

Future research may relax each limitation to better approximate the real version of Keeta. The model is meant to be owned and experimented with by the protocol engineers. The Quint model, or simply its methodology, can enable them to iteratively verify additions as the protocol evolves.

The two abstractions of the previous section are the natural starting point. Modeling dynamic weights and block expiry, and re-verifying safety and lockout-freedom, would confirm or dispel the concerns raised in Section 6.

Liveness is a larger undertaking. General progress under fairness or partial synchrony is out of scope, and the model is deliberately incompatible with it. The synchronized-clock treatment of expiry, which lets any temporary vote lapse in any order, is sound for safety but not for liveness, where the timing of expiries is what decides progress. A faithful liveness result would need a separate model of clocks and message delay, and is left to other work.

The generalization of safety is likewise only argued on paper. A machine-checked proof for every $n$ would follow from establishing `NoFork` as an inductive invariant, the route sketched in Section 5.3.

This work verifies one layer, consensus, and leaves the surrounding layers abstract. The application layer above it, spanning token operations, balances, access control, identity, and cross-network bridging, and the infrastructure layer below it, spanning peer-to-peer dissemination, storage, and transport, are each a candidate for the same treatment.

## 8 Conclusion

Overall, Keeta is an ambitious attempt at tackling the structural inefficiencies affecting contemporary financial infrastructure. It draws on advances from across computer science and finance, recombined from first principles into something that, taken as a whole, is genuinely new. Such a system invites study from many angles, yet so far it rests on a single whitepaper.

This paper explored the consensus protocol at the foundation of Keeta and established the safety of its consensus core under a fixed, equally weighted representative set, showing that agreement holds under the protocol's fault bound and, by a quorum-intersection argument, at every network size rather than only the configurations checked. It showed that a contended account can still be driven into a permanent lockout, and documented the liveness trade-offs of shared accounts that the whitepaper discussed only passingly. The specification it produced can assist the protocol's designers in safely iterating on new additions and modifications. Much of the knowledge around Keeta has existed only as word of mouth and informal conversation, and part of it is set down in writing here, to advance the general understanding of the protocol.

The paper opened with the cost and friction of moving money across borders, and introduced the promise that it could all be settled on a shared ledger, escaping the fragmented and capital-hungry correspondent banking system. That promise holds only if the ledger beneath it can scale to the demand, while maintaining trust. Establishing the safety of Keeta's consensus core is one step toward earning that trust. A robust, parallel, leaderless settlement layer, standing on a verified foundation, is the infrastructure global payments have long needed. Whether Keeta grows into it is a matter of time and adoption, but its foundation now stands on more solid ground.

## Disclaimer

The author has performed contractual work for Keeta Token Genesis LLC and otherwise has a vested interest in the success of the project. The paper was not part of said contractual work. The research was performed independently; contact with Keeta engineers was limited to factual clarification of the protocol, with no oversight of or involvement in the research.

The author used AI-powered search engines (Google AI mode, Perplexity, Claude Deep Research) to navigate prior literature, and the AI agent "Claude Code" assisted with the creation of Keeta.qnt and the writing of the paper. The author rigorously reviewed all content and takes full responsibility for it.

## References

[1] Karolos Antoniadis, Antoine Desjardins, Vincent Gramoli, Rachid Guerraoui, and Igor Zablotchi. 2021. Leaderless Consensus. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 392–402. https://doi.org/10.1109/ICDCS51616.2021.00045
Cited on page 7.

[2] Mathieu Baudet, George Danezis, and Alberto Sonnino. 2020. FastPay: High-Performance Byzantine Fault Tolerant Settlement. In *2nd ACM Conference on Advances in Financial Technologies (AFT '20)*. ACM, 163–177. https://doi.org/10.1145/3419614.3423249
Cited on pages 6 and 21.

[3] Ulrich Bindseil and George Pantelopoulos. 2022. *Towards the Holy Grail of Cross-Border Payments*. Working Paper Series 2693. European Central Bank. https://doi.org/10.2866/333725
Cited on page 3.

[4] Sam Blackshear, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Xun Li, Mark Logan, Ashok Menon, et al. 2024. Sui Lutris: A Blockchain Combining Broadcast and Consensus. In *2024 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM. arXiv:2310.18042.
Cited on pages 10 and 22.

[5] Sean Braithwaite, Ethan Buchman, Igor Konnov, Zarko Milosevic, Ilina Stoilkovska, Josef Widder, and Anca Zamfir. 2020. Formal Specification and Model Checking of the Tendermint Blockchain Synchronization Protocol. In *2nd International Workshop on Formal Methods for Blockchains (FMBC 2020) (OASIcs, Vol. 84)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 10:1–10:14. https://doi.org/10.4230/OASIcs.FMBC.2020.10
Cited on page 6.

[6] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI '99)*. 173–186.
Cited on pages 5 and 17.

[7] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, Dawn Song, and Roger Wattenhofer. 2016. On Scaling Decentralized Blockchains (A Position Paper). In *Financial Cryptography and Data Security (FC 2016) Workshops (Lecture Notes in Computer Science, Vol. 9604)*. Springer, 106–125.
Cited on page 7.

[8] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. 2020. Flash Boys 2.0: Frontrunning in Decentralized Exchanges, Miner Extractable Value, and Consensus Instability. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 910–927. arXiv:1904.05234.
Cited on page 7.

[9] Financial Stability Board. 2024. *Annual Progress Report on Meeting the Targets for Cross-border Payments: 2024 Report on Key Performance Indicators*. Technical Report. Financial Stability Board.
Cited on page 3.

[10] Roy Keene, Tanveer Wahid, Ezra Ripps, and Ty Schenk. 2025. *KeetaNet: Scalable Blockchain Banking*. Technical Report. Keeta.
Cited on pages 3, 4, 8, 14 and 17.

[11] V. Kukharenko, K. Ziborov, R. Sadykov, and R. Rezin. 2021. Verification of HotStuff BFT Consensus Protocol with TLA+/TLC in an Industrial Setting. In *Software Engineering and Algorithms (CSOC 2021) (Lecture Notes in Networks and Systems, Vol. 228)*. Springer, Cham. https://doi.org/10.1007/978-3-030-77448-6_9
Cited on page 6.

[12] Jure Kukovec, Daniel Cason, Igor Konnov, and Josef Widder. 2022. Specifying and Checking an Extension of Tendermint Consensus in TLA+. Presented at the TLA+ Community Event 2022.
Cited on page 6.

[13] Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems* 4, 3 (1982), 382–401. https://doi.org/10.1145/357172.357176
Cited on page 5.

[14] Colin LeMahieu. 2018. Nano: A Feeless Distributed Cryptocurrency Network. Whitepaper. https://content.nano.org/whitepaper/Nano_Whitepaper_en.pdf.
Cited on page 8.

[15] Pierre Tholoniat and Vincent Gramoli. 2019. Formal Verification of Blockchain Byzantine Fault Tolerance. arXiv preprint arXiv:1909.07453.
Cited on page 6.

[16] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. 1999. Model Checking TLA+ Specifications. In *Correct Hardware Design and Verification Methods (CHARME '99) (Lecture Notes in Computer Science, Vol. 1703)*. Springer, 54–66. https://doi.org/10.1007/3-540-48153-2_6
Cited on page 14.