Rinku: A Distributed Ledger For Mesh-Native Systems

rinkuchan.com

Abstract

Existing distributed ledgers assume persistent, well-connected infrastructure. They halt when the network partitions, require always-on RPC endpoints for state verification, and their smart contracts presume a synchronized world. These assumptions break in intermittently connected environments - ad-hoc mesh networks, field deployments with unreliable backhaul, maritime and remote operations, disaster response coordination, and autonomous agent fleets - where connectivity gaps are routine operating conditions, not exceptional failures.

Rinku is a DAG-based distributed ledger built around three primitives designed for these environments. Tunable consistency navigates the CAP tradeoff dynamically: delivering checkpoint finality when quorum is reachable, degrading gracefully to provisional availability during partitions, and deterministically reconciling divergent state when connectivity is restored. VerifiableObjects are self-contained, URL-encoded cryptographic proofs that carry everything needed for offline verification - no full node, no RPC endpoint, no network access. Any participant can receive a payment proof, verify it locally with a single BLS check, and act on it before ever touching external infrastructure. BYOP (Bring Your Own Proof) smart contracts accept VerifiableObjects as inputs, enabling contract logic to execute against proven external state without synchronous cross-chain or cross-contract calls - composable without centralized coordination.

Together, these primitives describe a ledger designed for environments where partitions are a first-class operating condition. The native RKU token has a hard cap of 30 million units with checkpoint-based emission and weighted proof-of-stake rewards.

Table of Contents

1. Introduction

Distributed ledgers have achieved remarkable success in environments where network connectivity is reliable and persistent. Bitcoin's Nakamoto consensus and Ethereum's Gasper protocol deliver strong probabilistic or deterministic finality under the assumption that a sufficient fraction of participants can communicate within reasonable time. BFT-family protocols - Tendermint, HotStuff, Bullshark, Mysticeti - push finality latency down to seconds or sub-seconds, but all share a fundamental constraint: they halt when the network cannot reach quorum.

This constraint is acceptable for data-center-class infrastructure where partitions are rare and short-lived. It is insufficient for a growing class of environments where intermittent connectivity is the norm: ad-hoc wireless mesh networks, mobile-first deployments in regions with unreliable backhaul, maritime and remote field operations, disaster response coordination, autonomous agent fleets, and industrial IoT meshes. These environments require the same economic

primitives - verifiable payments, enforceable contracts, trustless state sharing - that well-connected infrastructure readily supports.

The distributed systems literature offers rich solutions for partition tolerance - CRDTs, eventual consistency, Bayou-style conflict resolution, anti-entropy protocols - but these techniques have seen limited adoption in production blockchain design. The gap exists because financial state is not naturally commutative: transferring tokens is an inherently non-idempotent operation where order matters and conflicts have economic consequences.

Rinku bridges this gap. It provides a distributed ledger designed for environments where partitions are a first-class operating condition, not a side-effect to be circumnavigated. The protocol maintains strong finality when the network is connected and degrades gracefully to provisional operation during partitions, with a deterministic merge protocol that reconciles state when connectivity is restored.

1.1 The Partition Problem

Traditional blockchains prioritize a single canonical history with strong finality but sacrifice availability - the network halts when partitions prevent quorum. There are typically 3 categories for how these halts are handled:

- Category 1 (Nakamoto): Live during partition, reorg losers on reconnect, with no intelligent reconciliation
- Category 2 (BFT): Halt finalization, queue in mempool; safety over liveness
- Category 3 (Hybrid/Ethereum): Tips keep moving, finality pauses; halfway measure

So far, these have proven fairly sufficient for strongly connected infrastructure, but for networks that are disjoined or regularly fragmented there still remains to be a more appropriate solution.

1.2 Rinku's Position

Rinku takes a different position: tunable consistency. When the network is fully connected, Rinku Certified Consensus (RCC) delivers sub-second transaction finality through a dual-lane architecture: a Fast Lane for transfers and staking with immediate BLS-certified finality, and a Contract Lane for shared-state contract calls ordered through Quorum-Certified Checkpoints (QCC). When partitions occur, the network degrades gracefully to provisional acceptance - transactions continue locally, and state reconciles deterministically when partitions heal.

The core invariant: no honest user is prevented from transacting during a partition.
Naturally, the cost of this guarantee is that some transactions may be rolled back during merge if they conflict with transactions from other partitions. Intentional abuse is economically penalized.

1.3 VerifiableObjects as the User-Facing Primitive

Rinku's most distinctive user-facing concept is the VerifiableObject (VO): a self-contained, URL-encoded cryptographic proof that carries all data necessary for offline verification. Every meaningful output of the protocol - a payment confirmation, a contract execution receipt, a trust score attestation - is expressible as a portable rinku://vo/ URL. This collapses the distinction between "querying the chain" and "holding a proof" - any party with a VO can verify its claim without a full node, an RPC endpoint, or any network access at all. Section 3 expands on this design much further.

2. Trust Model & Terminology

Before describing the protocol, we establish the trust assumptions and define key terms precisely. These definitions apply throughout the paper.

2.1 Genesis Trust Anchor

The Rinku network is bootstrapped from a genesis checkpoint signed by an initial validator set defined in node configuration (GENESIS_VALIDATORS). This genesis checkpoint establishes:

- The initial account state (genesis allocation of 6,000,000 RKU)
- The founding validator set with initial stakes
- The root of the checkpoint hash chain

All subsequent state is cryptographically derived from this anchor. Non-genesis nodes verify the genesis validator set against their own configuration and will wipe and resync if the genesis state is inconsistent.

2.2 Validator Set Evolution

The validator set evolves through staking and unstaking transactions on the ledger:

Joining the validator set. Any account may become a validator by submitting a Stake transaction locking at least 100 RKU. Upon finalization (inclusion in a checkpoint signed by the current quorum), the staking account is added to the active validator set maintained by the ValidatorIdentityService. The new validator's BLS public key is registered and becomes eligible for checkpoint voting at the next height.

Leaving the validator set. A validator submits an Unstake transaction. The stake enters a 24-hour cooldown period during which it remains locked and slashable but no longer earns rewards. After cooldown, the stake is returned to the liquid balance. A 14-day unbonding

window applies for slashing purposes - evidence of misbehavior during the bonded period can still trigger slashing during unbonding.

Checkpoint signing. Each checkpoint is signed by the validators active at that height. The ConsensusService maintains a frozen snapshot of the validator set for each voting round, ensuring that staking changes mid-round do not affect quorum calculation. New validators participate in signing starting from the checkpoint following their stake finalization.

Genesis-to-runtime transition. The genesis node creates the initial validator accounts and registers their stakes. Non-genesis nodes bootstrap by syncing the genesis checkpoint and

adopting the genesis validator set from GENESIS_VALIDATORS configuration. During sync, if a node detects that its local genesis state is inconsistent with the peer's, it performs a full database wipe and resync to ensure all nodes share the same trust root. The runtime validator set evolves independently from the genesis set as staking transactions are processed.

2.3 Terminology: Acceptance vs. Finality

This paper uses precise terminology for transaction lifecycle states:

Term	Meaning	Guar
Submitted	Transaction received by a node and inserted into the DAG	No c
Fast-path finalized	Transaction has received fast-path BLS certificate from >2/3 active stake	Imme appl fast s ur
Finalized	Transaction referenced by a checkpoint signed by >2/3 total stake quorum	Irre majc
Provisionally finalized	Transaction referenced by a provisional checkpoint during partition mode	Vali subj merg
Reconciled	Previously provisional transaction that survived the merge protocol	Equi refe chec
Rolled back	Previously provisional	Remc

	transaction that was rejected during merge	stat audi

2.4 What Receipts Prove

Receipt Type	Mode	Anchor
Fast-path certificate	Normal	Batched BLS aggregate signature
Checkpoint receipt	Normal	BLS-signed checkpoint with state root
Provisional receipt	Partition	Provisional checkpoint with partition_epoch
Reconciliation receipt	Post-merge	Merge checkpoint with merge_report_has h
Rollback receipt	Post-merge	Merge report

2.5 Threat Model

Rinku's security properties rely on the following assumptions:

Honest majority. The protocol assumes that validators controlling more than 2/3 of total staked RKU follow the protocol honestly. This is the standard BFT assumption. Under this assumption, finalized checkpoints are irreversible - a conflicting checkpoint would require >1/3 stake to sign two different checkpoint hashes at the same height, which is detected and slashed (section 12).

Byzantine fault tolerance. Up to f < n/3 validators (by stake weight) may behave arbitrarily - equivocating, withholding votes, or broadcasting invalid messages. The protocol guarantees safety (no conflicting finalized checkpoints) as long as the honest majority assumption holds. Liveness requires >2/3 stake to be reachable for checkpoint finality; during partitions, liveness is maintained locally through provisional checkpoints (Section 8).

Partition attacker model. An adversary capable of partitioning the network can:

- Cause some partitions to operate under provisional finality (reduced but functional)
- Cause some transactions to be rolled back during merge if they conflict across partitions
- Trigger cascade rollbacks that affect innocent users whose transactions depended on rolled-back funds

An adversary cannot:

- Forge transactions (ECDSA P-256 signatures)
- Cause conflicting finalized (non-provisional) checkpoints (requires >2/3 honest stake)
- Double-spend without detection and penalty (nonce reuse is detected during merge; Section 12)
- Exploit partition tolerance for profit without incurring economic penalties that exceed the potential gain (Section 12.4)

Sybil resistance. Identity and influence in the protocol are derived from staked RKU. Weight calculations use sub-linear stake scaling (stake^0.5 * 2.0) to reduce the advantage of large stakers while maintaining Sybil resistance. Note: sub-linear scaling technically rewards stake splitting — N validators at stake S/N each yield aggregate weight proportional to S^0.5 * N^0.5, which grows with N. Sybil resistance is therefore load-bearing on the minimum stake requirement (MIN_STAKE = 100 RKU): creating N Sybil identities requires N × 100 RKU locked capital, limiting the practical splitting advantage. See Section 12.5 for detailed analysis.

3. VerifiableObject System

VerifiableObjects (VOs) are Rinku's universal container for portable, self-proving cryptographic claims. Every proof type in the system produces a rinku://vo/ URL with embedded proof data and freshness metadata. VOs are the primary interface between the Rinku protocol and external consumers - they are how the ledger communicates provable facts to the world. By

embedding all verification data within the proof itself, VOs eliminate the requirement for full-node infrastructure, RPC endpoints, or reliable network connectivity for state verification.

3.1 Proof Types

Type	Description	Use C
ContractOutput	StatefulReceipt with view keys, pre/post state roots, events	State
AccountProof	Balance, nonce, stake at a specific checkpoint	Accour
TxFinality	Transaction inclusion proof with Merkle path and BLS signature	Paymer
FastPathProof	Fast-path certificate with micro-checkpoint Merkle proof	Sub-se confir
WeightProof	Aggregate stake weight attestation	Trust anti-c
BatchProof	Multi-receipt verification with shared checkpoint context	Bulk v
StateWitness	Sparse Merkle multiproof for contract storage keys	Statel
Custom	Schema-defined proofs for application-specific claims	Extens

3.2 URL Encoding

VOs are serialized to JSON, DEFLATE-compressed, and encoded as URL-safe Base64:

rinku://vo/

Additional URI schemes for specific proof types:

- rinku://sp/ - Self-contained proofs (account state with full Merkle path to checkpoint)

- rinku://asp/ - Account state proofs (compact)

The URL encoding is designed so that a VO can be shared as a hyperlink, embedded in a QR code, or passed as a transaction parameter - collapsing the boundary between "data" and "proof of data."

3.3 Proof Freshness

Every VO carries optional ProofFreshness metadata:

- generated_at_checkpoint - checkpoint height at proof generation
- generated_at_timestamp - wall-clock time of generation
- max_age_checkpoints - optional expiry window

Verifiers can enforce proof age limits, preventing the use of stale proofs in time-sensitive operations. This is critical for BYOP transactions (Section 3.4) where a contract must ensure it is acting on recent state.

3.4 BYOP (Bring Your Own Proof) Transactions

Contracts accept ProofInput arrays as transaction parameters. Each proof input carries a VerifiableObject and a ProofExpectation specifying:

- Required proof type
- Chain ID (for potential cross-chain use)
- Minimum checkpoint height
- Expected state root

The runtime validates all proofs before contract execution begins and injects verified data into the WASM context under the proof.. namespace. This enables contracts to act as their own oracles - consuming proven facts from other contracts, accounts, or even external chains without synchronous state access.

Security against proof replay. Proof freshness requirements mitigate replay attacks where a valid but stale proof is resubmitted to a contract. The ProofExpectation includes a max_age_checkpoints field; the runtime computes the proof's age as current_checkpoint_height - generated_at_checkpoint and rejects proofs that exceed the caller's specified age limit. This enforcement occurs at the verifier's constraint level, not the proof's own max_age_checkpoints - ensuring that the consumer of a proof, not its producer, controls freshness requirements.

Cross-contract receipt composition. A contract can consume the StatefulReceipt of another contract's execution as a ProofInput. For example, a lending contract can accept a price oracle contract's receipt as proof of the current collateral value, without making a synchronous cross-contract call. The lending contract validates the oracle receipt's Merkle proof against the state root, checks freshness, and proceeds with the proven value. This pattern enables composability without re-entrancy risk.

4. Design Philosophy

4.1 URL-Native State

Rinku embeds the entire ledger state within cryptographically-linked URLs. Every proof, receipt, and state witness can be encoded as a self-contained URL that carries all data necessary for offline verification. This eliminates the dependency on full-node infrastructure for trust - any party holding a Rinku URL can independently verify its claim against a checkpoint anchor.

4.2 Self-Provable Ledger

Rinku's proof architecture differs fundamentally from SPV (Simplified Payment Verification) and traditional light client models:

SPV proofs (Bitcoin) prove transaction inclusion in a block via Merkle path but require the verifier to trust block headers, which in turn requires following the longest-chain rule. The verifier must either maintain a header chain or trust a third party that does. SPV proofs do not prove account state - only transaction inclusion.

Light clients (Ethereum Beacon Chain, Cosmos IBC) verify state transitions by tracking a committee of validators and their signatures. They require an ongoing connection to at least one honest full node to follow the validator set evolution. Without this connection, a light client's view becomes stale and unverifiable.

Rinku self-contained proofs carry everything needed for offline verification in a single URL:

1. The claim itself (account balance, transaction finality, contract output)
2. A Merkle inclusion proof connecting the claim to a checkpoint state root
3. The BLS aggregate signature over the checkpoint hash
4. A signer bitmap identifying which validators signed
5. Signer membership proofs (Merkle Sum Tree) proving each signer was a member of the validator set with their claimed stake weight

A verifier holding a self-contained proof and a trusted checkpoint (or the genesis checkpoint) can verify the claim without any network access, any RPC endpoint, or any trust in the proof's

provider. The proof is self-authenticating - either the math checks out or it doesn't.

This removes the reliance of "blockchain infrastructure" for read operations. Where Ethereum requires archive nodes, RPC providers (Infura, Alchemy), and client libraries to answer "what is Alice's balance?", Rinku answers the same question with a URL that anyone can verify with a quick, single cryptographic check.

4.3 Tunable Consistency

Rinku does not claim to "solve" or "beat" the CAP theorem. Instead, it navigates CAP by dynamically selecting the appropriate tradeoff:

- Normal operation: CP-like strong finality via dual-lane consensus. Fast Lane transactions (transfers, staking) achieve immediate finality through BLS-certified fast-path voting. All transactions achieve irreversible settlement through Quorum-Certified Checkpoints (QCC) backed by >2/3 total stake. This is not classical linearizability in the distributed systems sense, but provides the practical guarantees users expect: once finalized, a transaction cannot be reverted.
- Partition mode: Provisional availability. Transactions continue locally with explicitly labeled provisional finality. Clients are informed that these confirmations are subject to rollback.
- Post-partition: Deterministic convergence to global consistency through the merge protocol (Section 9). All nodes independently compute identical reconciled state from the same inputs.

The innovation is not the mode switching itself - it is the transaction taxonomy and reconciliation semantics that make this mode switching practical for financial state without unacceptable rollback risk.

4.4 CAP Analysis

Formally, Rinku's position in the CAP design space is:

Mode	Consistency	Availability
Normal	Strong (fast-path + QCC finality)	Full (fast-path cert + QCC checkpoint)
Partitioned	Provisional (local	Partial (Safe and

	consistency only)	BoundedSpend transactions only; CpOnly transactions are rejected)
Post-merge	Strong (deterministic reconciliation)	Temporarily reduced (merge computation)

Note: partition tolerance is a design property of the protocol, not a runtime state. Rinku is always partition-tolerant in the sense that partitions do not cause data loss or protocol-level inconsistency - the system's response to partitions is what varies between CP (Normal) and AP (Partitioned) behavior.

During partition, Rinku explicitly sacrifices global consistency in favor of local availability for eligible transaction types. The key insight is that this sacrifice is bounded and recoverable: the transaction taxonomy (Section 9.8) limits which operations can proceed, the partition budget system bounds the economic exposure, and the merge protocol deterministically recovers global consistency.

This is not eventual consistency in the CRDT sense - Rinku does not guarantee that all operations commute. Instead, it guarantees that non-commutative operations (financial transfers) are either merge-safe by design (within partition budget) or subject to explicit, deterministic conflict resolution with graduated economic penalties for abuse.

5. Network Architecture

5.1 Peer-to-Peer Layer

Rinku uses pure libp2p with gossipsub for all inter-node communication. There are no HTTP-based node-to-node calls. The protocol operates on two channels:

- Gossip topic (rinku/1.0.0): Transaction broadcast, checkpoint announcements, tip announcements, merge payloads
- Request-response protocol (/rinku/sync/1.0.0): Checkpoint sync, delta sync, snapshot recovery, presync, partition visibility queries

Note: 1.0.0 reflects current protocol version.

5.2 Node Roles

Validator nodes participate in consensus by staking RKU, voting on fast-path transaction certificates, signing QCC checkpoints, and proposing checkpoints when elected as leader. Validators run the full protocol stack including the partition detector, merge orchestrator, and tip consolidation service. Only the genesis node creates the initial validator accounts and registers their stakes; non-genesis validators adopt the validator set during sync.

Full nodes maintain a complete copy of the DAG and state but do not participate in checkpoint voting. They validate all transactions and checkpoints, serve API requests, relay gossip messages, and can independently verify the entire chain from genesis. Full nodes contribute to network resilience by increasing the number of honest relays.

Light clients (TBA..?). The self-contained proof architecture (Section 4.2) enables a lightweight verification mode where clients hold no ledger state. A light client can verify any claim by receiving a rinku://sp/ URL and checking the BLS signature, Merkle proof, and signer membership proofs against a trusted checkpoint anchor. This model requires no ongoing connection to any full node - verification is a single, stateless cryptographic operation.

5.3 Synchronization

Rinku implements a multi-mode synchronization system optimized for different scenarios:

Snapshot sync (new nodes). A joining node requests a complete state snapshot from a peer. The snapshot includes all account balances, stakes, contract state, and the current checkpoint chain. After applying the snapshot, the node replays any transactions in the DAG that occurred after the snapshot's checkpoint height. Transactions are only marked finalized: true if they appear in the snapshot's finalized_tx_hashes or in checkpoint finalized_tx_hashes lists - preventing spurious finality attribution during recovery.

Delta sync (catching up). A node that has been briefly disconnected requests only the transactions and checkpoints created since its last known checkpoint. This is a lightweight operation that avoids retransmitting the full state. Delta sync also reconciles the validator identity service, ensuring the catching-up node's total_active_stake matches the network.

Presync (quick bootstrap). Before performing a full snapshot sync, a joining node performs a lightweight handshake to determine the peer's checkpoint height and validator set. This allows the node to estimate the sync workload and select the most appropriate sync mode.

Persistent storage. All state is persisted using redb, a lightweight embedded database. The persistent transaction counter (total_transactions) is stored in metadata alongside gas_price, total_supply, and genesis_time, ensuring accurate counts survive DAG pruning across restarts.

Ghost account prevention. During sync, account push-back filters out stale accounts (zero balance, zero nonce) to prevent state contamination from obsolete data. Known edge case:

this heuristic incorrectly filters legitimate accounts that have been fully drained (sent their entire balance with no incoming transactions). Such accounts would need to receive funds again to reappear in the synced state. In practice, this affects only accounts with exactly zero balance and zero nonce — an account that ever transacted (nonce > 0) is preserved regardless of balance.

6. Consensus: Rinku Certified Consensus (RCC) — Dual-Lane

Architecture
Rinku implements a dual-lane consensus architecture called Rinku Certified Consensus (RCC). Transactions are classified into two lanes based on their state access pattern: the Fast Lane provides sub-second finality for transfers and single-owner operations through batched BLS certificates, while the Contract Lane queues shared-state contract calls for deterministic ordering through Quorum-Certified Checkpoints (QCC). Both lanes converge at the checkpoint layer, which provides the canonical anchor for proofs, pruning, and synchronization.

Intellectual lineage. RCC draws on the insight from the Avalanche protocol family [9] — that reliable broadcast with stake-weighted voting can achieve finality without rounds — but diverges in two key ways. First, RCC uses a single broadcast round where all validators participate in parallel (no repeated sub-sampled polling). Second, RCC restricts fast-path finality to operations with non-overlapping state access (transfers, staking), routing shared-state operations (contract calls) through checkpoint ordering. This lane separation eliminates the need for a speculative execution overlay: fast-path state changes are applied directly to canonical state.

6.1 Fast Lane: BLS-Certified Transaction Finality

The Fast Lane provides sub-second finality for eligible transaction types through stake-weighted validator voting and batched BLS certificate generation (observed median ~43ms, p95 ~200ms, p99 ~500ms; see Appendix C.2).

Eligible transaction types: All non-contract transaction types — Transfer, Stake, Unstake, ClaimRewards, Reward, DataOnly, Consolidation — are unconditionally Fast Lane eligible. Additionally, contract call transactions (calls to existing deployed contracts) are conditionally Fast Lane eligible through write-set conflict detection (Section 6.7). Contract deploy transactions are always routed to the Contract Lane. The lane classification for non-deploy contract calls is dynamic: a contract call enters the Fast Lane unless its write-set overlaps with another in-flight transaction's write-set, in which case it is deferred to checkpoint ordering.

The fast-path process:

1. Broadcast. When a validator receives a valid Fast Lane transaction (via API or gossip), it broadcasts a TxConfirmBroadcast message containing the full transaction, the validator's address, and its stake weight.
2. Voting. Every validator that receives the broadcast validates the transaction (signature, nonce, balance sufficiency) and responds with a TxConfirmAck containing its stake weight and BLS signature over the transaction hash.
3. Certificate formation. The originating node accumulates ACKs. When accumulated stake exceeds 2/3 of total validator stake, a FastPathCertificate is formed — a batched BLS aggregate signature over the transaction hashes, with a signer bitmap and Merkle root of the certified batch.
4. Canonical execution. Upon certification, state effects are applied directly to canonical state — balances are updated, nonces are incremented, gas fees are burned. There is no speculative overlay; the state change is immediate and canonical.

Key properties:

- Leaderless. No leader election is required for fast-path finality. All validators participate symmetrically in voting.
- Monotonic finality. Once a transaction receives a fast-path certificate backed by >2/3 stake, it is final. There is no subsequent phase that can revert it (under the honest majority assumption).
- Direct canonical state application. State is applied directly to the canonical account map at certification time, not deferred to a speculative overlay. This eliminates overlay replay costs and the class of bugs associated with speculative-to-canonical promotion.
- Nonce-ordered execution. Fast-path-confirmed transactions with future nonces (higher than the account's current nonce) are deferred and retried in (from_address, nonce) sorted order, ensuring sequential execution unblocks dependent transactions without rejection.
- Bounded tracking. Fast-path finalized transactions are tracked in a fast_path_finalized_txs map (capped at MAX_FAST_PATH_POOL = 5,000 entries) to prevent double-execution during checkpoint finalization. The map is cleared at each checkpoint boundary.

6.2 Contract Lane: Checkpoint-Ordered Execution

Contract deploy transactions and contract call transactions whose write-sets conflict with in-flight fast-path transactions are routed to the Contract Lane for deterministic ordering through the checkpoint process.

Rationale. Contract calls may read and write shared storage keys. Two concurrent contract calls that access the same storage cannot be safely finalized independently — execution order

determines the final state. For conflicting contract calls (overlapping write-sets) and all deploy transactions, routing through checkpoint ordering provides a total order over all contract executions within each checkpoint interval, ensuring deterministic state transitions across all nodes. Non-conflicting contract calls are eligible for fast-path finality via write-set conflict detection (Section 6.7).

Contract Lane flow:

1. Contract transaction is received, validated (signature, nonce, gas budget), and inserted into the DAG. For contract calls (not deploys), the transaction is first evaluated for fast-path eligibility via write-set conflict detection.
2. If the transaction is fast-path eligible and does not conflict, gas is deducted via fast-path and the transaction is tracked as a fast-path special transaction for WASM execution at checkpoint time.
3. At checkpoint time, the leader collects all unfinalized contract transactions (plus any Fast Lane transactions not yet checkpoint-finalized). Fast-path confirmed contract calls have their WASM executed during checkpoint finalization (gas already deducted by fast-path; execution gas charged separately).
4. Transactions are ordered deterministically (Section 7) and executed in batch, with state root computation after all executions complete.
5. The resulting checkpoint includes the contract execution results in its finalized_tx_hashes and state root.

6.3 Quorum-Certified Checkpoints (QCC)

QCC checkpoints are the canonical anchoring mechanism for both lanes. They serve as sync points for new nodes, enable DAG pruning, and provide the checkpoint anchors used for proof generation.

QCC creation flow:

1. A leader is elected for each checkpoint height using the deterministic stake-weighted mechanism described in Section 6.5.

2. The leader collects all unfinalized transactions since the last checkpoint from its local DAG, including both fast-path finalized transactions and Contract Lane transactions.

3. Transactions are filtered using a PROPAGATION_GRACE_MS window (default 5,000ms) to ensure sufficient time for gossip propagation.

4. The leader simulates execution: Fast Lane transactions already applied via fast-path are skipped (state effects already canonical); Contract Lane transactions are executed in order; state root is computed from the Sparse Merkle Trie.

5. The leader creates a checkpoint proposal containing tx_merkle_root, state_root, and finalized_tx_hashes, signed with its BLS key.

6. Vote collection. The leader broadcasts the checkpoint proposal via CheckpointAnnouncement and actively solicits votes from validators via both gossipsub and direct request-response. Each validator verifies the checkpoint (validates state root against local state, verifies transaction inclusion) and returns a BLS signature vote.

7. Quorum certification. When the leader accumulates BLS votes from validators controlling ≥2/3 of total stake, it aggregates the signatures into a single BLS aggregate signature and broadcasts the certified checkpoint. This is the QCC — a Quorum-Certified Checkpoint.

Key properties:

- BLS quorum required. Every checkpoint must carry an aggregate BLS signature backed by ≥2/3 of total validator stake. Single-proposer-signed checkpoints are not accepted (except during genesis bootstrap).
- Parallel vote solicitation. The leader uses both gossipsub broadcast and direct P2P request-response in parallel to collect votes, with a configurable deadline (PARALLEL_QCC_DEADLINE_MS = 4,000ms). This dual-path approach ensures vote collection completes even if gossip delivery is delayed.
- Non-blocking. Checkpoint creation never blocks fast-path transaction processing. If the leader is offline or slow, fast-path transactions continue to achieve finality via the Fast Lane.
- Proof-stripped gossip. Checkpoint announcements via gossipsub do not include precomputed Merkle proofs (which can exceed the 2MB gossipsub message limit at scale). Proofs are delivered exclusively via direct P2P SYNC-PUSH after quorum certification.
- Enabling proofs. QCC checkpoints provide the anchors referenced by VerifiableObjects (Section 3), Merkle inclusion proofs, and finality certificates.
- DAG pruning. Transactions included in a QCC checkpoint are pruned from the in-memory DAG, keeping the working set bounded.
- Fast-path tracking cleanup. After checkpoint finalization, the fast_path_finalized_txs tracking map is cleared, resetting the bounded pool for the next checkpoint interval.

6.3.1 Failure Modes

Scenario Impact Recovery

Minority validators offline Fast-path still achieves 2/3 if Offline validators sync via

	remaining stake suffices; QCC quorum may take longer	delta recor
Scenario	Impact	Rec
Leader offline	No QCC checkpoint created at this height; fast-path transactions continue to finalize	Lea act ele
State root divergence	Validator detects mismatch between checkpoint state root and local state	Del res reb
Network partition	Neither fast-path nor QCC can reach 2/3 global stake	Par (Se acc par

6.3.2 Consensus-Execution Separation

Non-proposer nodes do not re-execute all transactions during checkpoint application. Instead, they verify the SMT (Sparse Merkle Trie) proofs included in the checkpoint against the checkpoint's state root and apply account states directly. This significantly reduces write-lock hold times on non-proposer nodes, as they only need to verify proofs and update account states rather than replay all transaction logic.

If proof verification fails (state root mismatch), the node falls back to full execution. If the resulting state root still doesn't match, the node rolls back to the pre-checkpoint state and triggers a full sync.

6.4 BLS Aggregate Signatures

Rinku uses the BLS12-381 signature scheme (via the blst library, min_pk variant) for checkpoint signing:

Signature scheme. Signatures are on the G2 group (96 bytes compressed); public keys are on G1 (48 bytes compressed). Each validator signs the checkpoint hash with their BLS secret key, producing an individual signature.

Aggregation. Individual validator signatures are aggregated into a single 96-byte aggregate signature using AggregateSignature::aggregate. This aggregation is additive - the

aggregate signature can be verified against the aggregate of the corresponding public keys in a single pairing check, regardless of the number of signers.

Signer bitmaps. To identify which validators contributed to an aggregate signature without transmitting the full validator list, Rinku uses a compact bitfield. The signer_bitmap is a

Vec where the i-th bit corresponds to the i-th validator in the deterministically sorted validator set. A set bit indicates that validator signed the checkpoint. This enables any verifier with knowledge of the validator set to reconstruct the aggregate public key and verify the aggregate signature.

Verification. Given a checkpoint hash, aggregate signature, signer bitmap, and validator set, verification proceeds: (1) parse the bitmap to identify signers, (2) compute the signer stake as the sum of effective stakes of all signers and check that signer_stake / total_stake ≥ 2/3 (stake-weighted quorum, not signer-count quorum), (3) aggregate the signer public keys, (4) verify the aggregate signature against the aggregate public key and checkpoint hash. This is a constant-time operation regardless of the number of signers (a single pairing check). Verification is enforced at checkpoint reception — both the gossip handler (CheckpointAnnouncement) and the leader fallback path reject checkpoints that fail BLS verification. Checkpoints that lack an aggregate signature (e.g., during initial genesis bootstrap or sync of legacy checkpoints) are accepted with a warning log but not rejected, allowing gradual rollout of BLS enforcement.

Rogue-key mitigation. BLS aggregate signatures are vulnerable to rogue public key attacks where an adversary constructs a public key that cancels another validator's key during aggregation, allowing forgery of aggregate signatures. Rinku mitigates this by using the min_pk variant of BLS12-381 (public keys on G1, signatures on G2) and requiring proof-of-knowledge (PoK) of the BLS secret key at validator registration: each validator must produce a valid BLS signature over a registration message containing their address and stake transaction hash. This signature serves as a PoK and is verified before the validator's BLS public key is added to the active validator set. This follows the construction described by Boneh, Drijvers, and Neven [5] for secure multi-signature aggregation without the need for a deaggregation step.

Double-sign detection. The ConsensusService monitors for validators that sign two different checkpoint hashes at the same height. Double-signing triggers an immediate 15% stake slash and addition to a slashed_validators set, which reduces the validator's voting power in all pending rounds.

6.5 Leader Election

Rinku uses a deterministic, stake-weighted leader election mechanism based on verifiable randomness:

Randomness derivation. For each checkpoint height, a 32-byte seed is computed as:

```
randomness = SHA-256("RINKU_LEADER_ELECTION_V1" || checkpoint_height || previous_checkpoint_hash)
```

Since all validators in consensus share the same previous_checkpoint_hash and target height, they all derive identical randomness. The seed is unpredictable before the previous checkpoint is finalized (depends on the previous checkpoint's hash) but deterministic afterward.

Stake-weighted selection. The active validator set is sorted by address (ensuring identical ordering on all nodes). The randomness is mapped to a target value in the range [0, total_stake). The algorithm iterates through the sorted validators, accumulating stake until the cumulative sum exceeds the target. The validator that crosses the threshold is elected leader. Over time, each validator's probability of election is proportional to their stake.

Rotation. Because checkpoint_height is an input to the randomness, the leader changes every checkpoint. The combination of height-based rotation and stake-weighted selection ensures both fairness (proportional to stake) and unpredictability (cannot be known until the previous checkpoint finalizes).

Liveness fallback. If the elected leader fails to produce a checkpoint within a configurable timeout (leader_timeout_ms, default 45 seconds), a fallback mechanism activates. The should_fallback function uses a modified height input (checkpoint_height + 1000000) for randomness, effectively electing an emergency replacement leader. This ensures liveness even if the primary leader is offline. Clock skew consideration: different validators may trigger the fallback at slightly different wall-clock times due to clock skew, potentially resulting in a brief window where both the primary and fallback leaders are producing checkpoints simultaneously. This is safe — duplicate checkpoint proposals at the same height are resolved by the BLS quorum mechanism (only one can achieve 2/3 stake), and the other is discarded. However, clock skew increases the time to finality during leader failure by up to the skew magnitude.

6.6 Relationship Between Fast Lane and QCC Checkpoints

Property	Fast Lane	QCC Che
Latency	~43ms median, ~500ms p99	~5-10s
Quorum	>2/3 total validator stake (TxConfirmAck votes)	>2/3 to (BLS ch
Stake basis	Total validator stake	Total v

Eligible TXs	All non-contract types + non-conflicting contract calls (Section 6.7)	All tra (includ
State application	Immediate upon certification (canonical); contract WASM deferred to checkpoint	Fast-p Contra
Irreversibility	Irreversible under honest majority	Irreve aggreg
Proof artifact	FastPathProof via micro-checkpoint (Section 6.8)	QCC ch root,
Proof availability	~200-500ms (micro-checkpoint cycle)	After (~5-10

Quorum stake basis. Both fast-path and QCC quorum are measured against total validator stake (the sum of all registered validators' stakes, whether online or offline). This means both lanes require >2/3 of total stake to be reachable and responsive for finality. If the reachable stake drops below this threshold, the partition detector (Section 8) activates and the network transitions to provisional mode. Using total stake (rather than active stake) as the denominator provides a stronger safety guarantee: fast-path certification is backed by the same absolute stake threshold as QCC checkpoints.

6.7 Write-Set Conflict Detection & Contract Fast-Path

To enable sub-second finality for non-conflicting contract calls, Rinku implements a write-set based conflict detection system that dynamically determines whether a contract call can safely use the Fast Lane.

Write-set computation. Every fast-path executed transaction produces a WriteSet — a deterministic summary of the state keys it modifies. The write-set contains entries of the form (key, value_hash) where key identifies the state resource (account address or contract::state for contract storage) and value_hash is the SHA-256 hash of the serialized post-mutation value. The write-set is canonicalized by sorting entries lexicographically by key and computing a deterministic hash over the sorted (key, value_hash) pairs. Write-set hashes are propagated through all fast-path gossip messages — TxConfirmBroadcast, TxConfirmAck, FastPathAck, and FastPathFinality — enabling validators to verify write-set consistency across the voting round.

Conflict tracker. The WriteSetConflictTracker maintains a map of in-flight state keys to the transaction hashes currently modifying them. When a new transaction is submitted for fast-path execution:

1. The tracker checks whether any key in the transaction's write-set is already claimed by an in-flight transaction.
2. If no overlap exists, the transaction is registered in the tracker and proceeds through fast-path.
3. If any key overlaps (conflict detected), the transaction is rolled back from fast-path state and deferred to the Contract Lane for checkpoint ordering.

The tracker is cleared at each checkpoint boundary (both proposer and non-proposer paths), resetting the conflict detection window for the next checkpoint interval. This ensures that conflict tracking does not accumulate stale state across checkpoints.

Contract call eligibility. Only contract call transactions (calls to existing deployed contracts) are eligible for fast-path. Contract deploy transactions are always routed to the Contract Lane because deployment creates new shared state that cannot be conflict-checked against prior write-sets. The eligibility check parses the contract transaction data to distinguish Call variants from Deploy variants.

Gas handling for fast-path contracts. When a contract call achieves fast-path confirmation, base gas is deducted immediately during fast-path execution. WASM execution is deferred to checkpoint finalization time, where the contract runtime charges execution gas separately. This two-phase gas model ensures that fast-path gas deduction is reversible if the transaction is later rolled back due to a conflict detected by another validator.

6.8 Micro-Checkpoints

Micro-checkpoints provide sub-second Sparse Merkle Trie (SMT) state snapshots between QCC checkpoints, enabling near-instant proof generation for fast-path confirmed transactions without waiting for the next checkpoint cycle.

Architecture. The MicroCheckpointService runs as a background tokio task alongside the fast-path batch processor. Every 200ms, it processes pending write-sets from recently confirmed fast-path transactions and produces a micro-checkpoint — a lightweight SMT state snapshot containing the Merkle roots for accounts modified since the last micro-checkpoint.

Data structures:

- Micro-checkpoint ring buffer. A fixed-capacity ring buffer of MAX_MICRO_CHECKPOINTS (default 64) entries. Each entry contains a monotonically increasing sequence number, the SMT state root at that point, and a timestamp. When the buffer is full, the oldest entry is evicted.

- Pending write-set queue. Fast-path confirmed transactions enqueue their write-sets here. The background task drains this queue every 200ms and applies the state changes to the micro-checkpoint SMT.
- Transaction-to-micro-checkpoint index. A map from transaction hash to the micro-checkpoint sequence number that includes its state, enabling O(1) lookup of the relevant micro-checkpoint for proof generation.

Proof generation. When a client requests a proof for a fast-path confirmed transaction (via GET /api/tx/:hash/fast-path-proof), the node:

1. Looks up the micro-checkpoint sequence for the transaction hash.
2. Retrieves the micro-checkpoint's SMT state root.
3. Generates a Merkle inclusion proof from the micro-checkpoint SMT.
4. Returns a FastPathProof VerifiableObject containing the proof, the micro-checkpoint state root, the fast-path certificate, and proof freshness metadata.

Lifecycle. Micro-checkpoint state is ephemeral — it is not persisted to disk and is cleared when a QCC checkpoint is finalized (via on_qcc_checkpoint). After QCC finalization, the canonical checkpoint proofs supersede micro-checkpoint proofs. The micro-checkpoint index is pruned of entries for transactions that are now included in the QCC checkpoint.

Relationship to QCC proofs. Micro-checkpoint proofs and QCC checkpoint proofs serve different roles:

Property	Micro-Checkpoint Proof	QCC Chec
Availability	~200ms after fast-path confirmation	After ne (~5-10s)
Anchor	Micro-checkpoint SMT state root	QCC chec with BLS
Durability	Ephemeral (ring buffer, cleared at checkpoint)	Permaner checkpoi
Security	Backed by fast-path certificate (>2/3 stake)	Backed b BLS sigr
Use case	Immediate payment confirmation, real-time UX	Settleme verifica

Applications requiring immediate proof of payment can use micro-checkpoint proofs (available sub-second), then optionally upgrade to QCC checkpoint proofs for archival or cross-system

verification.

6.9 View Change Protocol

Rinku implements a formal view change protocol for leader election fault tolerance, replacing informal leader timeout handling with a structured voting mechanism.

When the elected leader for a checkpoint height fails to produce a checkpoint within the LEADER_TIMEOUT window (default 45 seconds), validators initiate a view change:

1. Timeout vote. Each validator that detects a leader timeout broadcasts a LeaderTimeout vote containing the checkpoint height and the validator's BLS signature.
2. View change quorum. When a node accumulates LeaderTimeout votes from validators controlling ≥2/3 of total stake, a view change is triggered.
3. Fallback leader election. The fallback leader is deterministically elected using a modified height input (checkpoint_height + 1000000) for randomness derivation, ensuring all validators agree on the replacement leader.
4. Checkpoint production. The fallback leader produces a checkpoint using the same QCC flow (Section 6.3), including vote collection and BLS aggregation.

The formal protocol ensures that view changes are coordinated across the validator set rather than triggered by individual node timeouts, preventing split-brain scenarios where multiple nodes attempt to act as fallback leader simultaneously.

6.10 Write-Ahead Log (WAL)

Rinku implements a Write-Ahead Log for crash recovery, ensuring state consistency across unexpected node restarts.

Design. Before any state mutation (account balance update, nonce increment, contract storage write), the mutation is serialized and appended to the WAL. The WAL entry contains the transaction hash, the mutation type, the pre-mutation and post-mutation values, and a monotonic sequence number. Only after the WAL entry is durably written (fsynced) does the state mutation proceed.

Recovery. On node startup, the WAL is scanned for incomplete mutations — entries that were written but whose corresponding state updates may not have been flushed to the redb persistent store. Incomplete mutations are replayed in sequence order, bringing the persistent state to a consistent point. After recovery, the WAL is truncated.

Checkpoint integration. WAL entries are truncated up to the last finalized checkpoint height. Since checkpoints represent durably persisted state snapshots, all mutations prior to the

checkpoint are guaranteed to be reflected in the persistent store.

6.11 Data Availability Layer

Rinku separates consensus and data availability to support large transaction payloads without bloating the consensus-critical message path.

Erasure coding. Large transaction body payloads (those exceeding the inline gossipsub threshold) are DEFLATE-compressed and then Reed-Solomon erasure-coded. The erasure coding produces n data shards and k parity shards, where any n of the n + k total shards are sufficient to reconstruct the original payload. This provides data availability guarantees even when up to k shards are lost or withheld.

Separation from consensus. Checkpoint announcements via gossipsub contain only the transaction hashes, state root, and BLS signatures — not the full transaction bodies. This keeps consensus messages within gossipsub size limits (2MB). Full transaction data is available via:

1. Direct P2P request-response (/rinku/sync/1.0.0): Nodes can request missing transaction bodies by hash.
2. Erasure-coded shard distribution: For large payloads, shards are distributed across peers, and any sufficient subset can reconstruct the full payload.

This design ensures that consensus (agreeing on the set of finalized transactions) proceeds independently of data availability (ensuring all nodes can access the full transaction data), preventing large contract payloads from bottlenecking checkpoint finalization.

7. DAG Structure & Transaction Ordering

7.1 Transaction DAG

Transactions in Rinku form a DAG rather than a linear chain. Each transaction references one or more parent transactions (DAG tips at the time of submission), creating a partial order that enables parallel processing.

7.2 Tip Selection

Rinku uses a Sparse DAG Sampling algorithm to prevent tip explosion while maintaining DAG connectivity and Sybil resistance.

MAX_SAMPLED_TIPS. The maximum number of parent references a transaction selects is bounded at 16. This prevents the DAG from growing excessively wide while maintaining sufficient connectivity for parallel validation.

Weighted selection with diversity. When the number of available tips exceeds 16, the sampling algorithm splits selection into two halves:

1. Guaranteed selection (top 8): The 8 tips with the highest sender account weight are always included. This ensures that well-staked, high-reputation transactions are preferentially referenced, providing Sybil resistance - an attacker flooding the network with low-weight transactions cannot crowd out legitimate tips.
2. Random sampling (bottom 8): The remaining 8 slots are filled by random sampling from all other available tips. This maintains DAG diversity, prevents the graph from narrowing to a single chain of high-weight transactions, and ensures that transactions from lower-weight (but honest) participants are eventually incorporated.

Tip consolidation. A background TipConsolidator service runs on validator nodes. When the tip count exceeds a threshold (default 100), it enters aggressive consolidation mode, periodically creating Consolidation transactions that reference 16 tips at once. These anchor transactions merge divergent branches back into fewer tips, keeping the DAG's working set manageable.

Orphan parent handling. If a transaction arrives with parent references to hashes not found in the local DAG (orphan parents), the node automatically injects current known tips as parents. This ensures the transaction attaches to the main graph even if some referenced parents were pruned or never received.

7.3 DAG Weight Calculation

Transaction weight in the DAG context (distinct from WPoS reward weight in Section 11.3) determines transaction ordering priority, tip selection preference, and conflict resolution outcomes. It is computed as:

effective_weight = (age_weight * balance_weight + stake_weight) * (1.0 - reputation_penalty)

Where:

- age_weight: Time-based component since transaction insertion
- balance_weight: Derived from the sender's account balance
- stake_weight: Sub-linear bonus from staked tokens, computed as stake^0.5 * 2.0. The square-root scaling reduces the advantage of large stakers - doubling stake increases weight by only ~41%, not 100%. This provides meaningful Sybil resistance while limiting plutocratic concentration.
- reputation_penalty: A value in [0.0, 1.0] reflecting accumulated partition-tolerance violations (see Section 12). An account with a 0.50 reputation penalty has its effective weight halved in all weight calculations.

The sub-linear stake weight is a deliberate design choice for decentralization: it makes it more capital-efficient to distribute stake across multiple honest validators than to concentrate it in a single large validator, while still ensuring that staked participants have significantly more influence than unstaked ones.

7.4 Cumulative Weight & Conflict Resolution

Intra-partition fork resolution. Within a single connected network (no partition), DAG forks are resolved by cumulative weight. The cumulative weight of a transaction is the sum of its own weight plus the weights of all transactions that directly or transitively reference it as a parent. When two transactions conflict (same sender, same nonce), the transaction with higher cumulative weight in the DAG is preferred. This mechanism provides probabilistic convergence similar to Bitcoin's longest-chain rule, but using stake-weighted attestation rather than proof-of-work.

Cross-partition conflict resolution. When partitions heal and the merge protocol runs (Section 9), conflict resolution uses a more sophisticated multi-factor algorithm that considers cumulative weight, visible stake percentage, and lexicographic hash tiebreaking. This is necessary because cumulative weight alone is not a fair comparison across partitions of different sizes - a transaction in a 60%-stake partition would naturally accumulate more weight than a transaction in a 40%-stake partition, even if both are equally valid.

8. Partition Tolerance

Consider any network where sub-groups regularly lose connectivity for 30-120 seconds: a fleet of field devices on an unreliable mesh, a maritime vessel cluster with intermittent satellite uplink, or a disaster response team operating on degraded infrastructure. In these environments, partition mode is the expected operating state, not an edge case. The transaction classification system ensures that data-only operations (Safe) continue uninterrupted, bounded financial transactions (BoundedSpend) proceed within pre-configured risk limits, and consensus-critical operations like staking changes (CpOnly) wait for full network reconnection.

8.1 Detection

Rinku implements a three-state partition detector that continuously monitors network health:

NORMAL ──[visible stake < 2/3]──► SUSPECTED ──[timeout T_conf]──► PARTITIONED

```
└──────────────[visible stake ≥ 2/3 for T_recovery]──────────────────────
```

Parameter	Default	Descr
T_conf	30s	Confi decla
T_recovery	10s	Quoru befor
Stake visibility threshold	66.66%	Minim norma

The PartitionDetector runs every 5 seconds, computing the percentage of total stake reachable via healthy gossip peers. Two detection signals are used:

1. Visible stake percentage. The primary signal. The detector queries the gossip service for currently connected peers, maps them to their validator identities, and sums their stake. If the sum falls below 2/3 of total stake, the detector transitions to SUSPECTED.
2. Checkpoint stall detection. A secondary signal. If no new checkpoint has been finalized for 3x the expected checkpoint interval and the node has unfinalized transactions, this indicates a likely quorum failure. This signal catches cases where visible stake calculations are stale due to peer-list update delays.

The SUSPECTED state serves as a damping buffer - transient network hiccups (brief disconnections, route changes) do not trigger partition mode unless they persist for T_conf seconds. This prevents unnecessary mode switching in mildly unstable networks.

8.2 Partition Epochs

Upon entering PARTITIONED mode, a monotonically increasing partition_epoch is assigned. This epoch tags all transactions and provisional checkpoints created during the partition, enabling the merge protocol to distinguish pre-partition from during-partition state.

8.3 Provisional Checkpoints

During a partition, the checkpoint quorum threshold relaxes from 2/3 of total stake to 2/3 of visible stake (minimum 1/3 of total stake as a safety floor). Checkpoints created under this relaxed quorum carry:

- provisional: true
- partition_epoch:
- visible_stake_pct:

Transactions finalized by provisional checkpoints receive provisional finality (see Section 2.3). They are treated as finalized for local operations but remain rollback-eligible during merge. Clients and applications are explicitly informed of the provisional status.

The safety floor of 1/3 total stake prevents a single isolated node from creating provisional checkpoints unilaterally - there must be a meaningful fraction of the validator set present in the partition for provisional operation to proceed.

8.4 Local Operation During Partition

All core operations continue during a partition, subject to the transaction classification system (Section 9.8):

- Safe transactions (DataOnly, Consolidation, Reward): Proceed without restriction. These are inherently merge-safe.
- BoundedSpend transactions (Transfer, Contract): Proceed up to the account's partition budget limit, if one is configured. Without a budget, they proceed without restriction but carry rollback risk.
- CpOnly transactions (Stake, Unstake, ClaimRewards): Rejected during partition. These operations modify the validator set or consensus-critical state and require full quorum for safety.

The partition is transparent to the application layer except for finality semantics (provisional vs. final) and CpOnly rejection. DAG tip selection, weight calculation, gas pricing, and smart contract execution operate identically in both modes.

8.5 Provisional Receipt Semantics

During partition mode, all receipts and VerifiableObjects carry the partition_epoch and provisional: true markers. A ProofFreshness attached to any VO generated during partition mode includes the partition epoch, enabling downstream consumers to make informed trust decisions. Applications that require settlement finality can choose to wait for post-merge

reconciliation; applications that prioritize responsiveness (messaging, social features) can accept provisional receipts immediately.

9. Reconciliation Semantics & Transaction Taxonomy

This section describes the core protocol innovation: a deterministic 5-phase merge protocol that reconciles divergent partition state while preserving maximum valid work from all partitions. This section is intended to be the most rigorous in the paper and will be the primary target of formal analysis.

Scope assumption: two-partition merge. The current merge protocol is designed and analyzed for the two-partition case: a single network split that produces two independent partition groups, which later reconnect and reconcile. Three-way or multi-way partitions (where the network fragments into three or more independent groups simultaneously) require a fundamentally different merge coordination protocol — pairwise merge of three independent provisional chains may not commute, and the merge order could affect the final state. For the current protocol version, multi-way partitions are handled by sequential pairwise merges (each reconnecting pair merges independently, with subsequent merges treating the merged state as the new baseline). Formal analysis of commutativity guarantees for sequential pairwise merge, and a native multi-party merge protocol, are future work.

9.1 Determinism Requirements

All merge computation must produce identical results on every node given the same inputs. This is the foundational constraint - without it, nodes would diverge after merge, which is worse than the partition itself.

Integer accounting: All balance operations during merge use u64 micro-units (1 RKU = 10^8 micro-RKU). Balances are converted from f64 to micro-units at merge entry and converted back only after reconciliation is complete. This eliminates floating-point nondeterminism entirely within the merge path.

Canonical ordering: Transactions are totally ordered by:

1. Per-account nonce ascending
2. DAG depth from fork-point (topological distance)
3. Transaction hash lexicographic ascending (deterministic tiebreaker)

This is a strict total order - no two transactions can be "equal" because transaction hashes are unique. Every node computing this order on the same transaction set produces the identical sequence.

Contract isolation: Contract calls are not re-executed during merge. The merge algorithm operates only on the balance transfers recorded in the original transaction execution (amount + gas fee). This avoids nondeterminism from contract execution order, memory state, or environmental dependencies.

Contract storage conflicts are resolved separately via "last-write-wins by weight" on individual storage keys. For each key written by transactions in both partitions, the write from the transaction with higher cumulative weight wins.

Known limitation: atomic multi-key updates. The per-key "last-write-wins" rule may produce inconsistent contract state when contract logic depends on multiple storage keys being written atomically. For example, if a contract maintains an invariant balance_a + balance_b ==

total and Partition A updates balance_a while Partition B updates balance_b, the merged state may violate the invariant. Mitigation strategies include contract-level merge hooks (Section 15.4), storage namespacing by partition safety class, and restricting AP-mode contract calls to safe subsets via the transaction classification system.

9.2 Phase 1 - DAG Exchange

Both partitions exchange their PartitionDAGDelta containing all transactions, provisional checkpoints, and DAG edges created since the fork point (last common confirmed checkpoint). After exchange, both sides have the complete merged transaction set.

Exchange is performed via gossip protocol messages (GossipMessage::MergePayload and GossipMessage::MergeResult). The merge payload includes a MergeRequest containing:

- All transactions created during the partition epoch
- Account state snapshots from the partition
- Provisional checkpoint chain
- Fork point reference (last common confirmed checkpoint height and hash)

9.3 Phase 2 - Conflict Detection

The merged transaction set is scanned for two conflict types:

Type 1 - Direct Double-Spend: Same account + same nonce in both partitions. This requires the user to have deliberately crafted a conflicting transaction - it cannot happen accidentally because nonces are sequential. Unambiguous evidence of intentional double-spending.

Type 2 - Economic Overdraft: No nonce collision, but combined spending from both partitions exceeds the account's pre-partition balance. Example: Alice had 100 RKU at the fork point, spent 60 RKU (nonce 5) in Partition A and 60 RKU (nonce 5 in Partition B, which was locally

valid because Partition B never saw nonce 5 from Partition A). Combined: 120 spent on 100 balance.

This may be innocent (the user didn't know about the partition and transacted on both sides) or opportunistic.

Detection algorithm:

1. Build a set of all (account, nonce) pairs across both partitions. Flag any duplicates as Type 1.
2. For each account that transacted in both partitions, compute pre-partition balance (from fork-point checkpoint), total sent and received in each partition. If combined net spend exceeds pre-partition balance, flag as Type 2.

9.4 Phase 3 - Weight Resolution

For each conflict, a deterministic winner is selected:

Direct double-spends: Winner is the transaction with higher cumulative DAG weight in the merged graph. If weights are within the WEIGHT_PROXIMITY threshold (1.5x), a tiebreaker applies. Formally, two weights W_a and W_b are considered "proximate" when max(W_a, W_b) / min(W_a, W_b) < 1.5 — i.e., neither weight exceeds 1.5 times the other. When weights are proximate, the tiebreaker is the partition with higher visible_stake_pct at the time of the provisional checkpoint that finalized the transaction. If still tied, the transaction with the lexicographically lower hash wins.

Economic overdrafts: All transactions from the conflicting account are ordered by nonce ascending, then cumulative weight descending. Starting from the pre-partition balance (in micro-units), transactions are replayed in this order. The first transaction that would cause an overdraft - and all subsequent transactions from that account - are marked as losers.

9.5 Phase 4 - Cascade Rollback

The key insight: rejecting a transaction doesn't just affect its sender - it affects everyone who received funds from that transaction and subsequently spent them. The cascade rollback algorithm traces these economic dependency chains.

Algorithm (pseudocode):

```
function cascade_rollback(losing_txs, all_txs):

  balances = snapshot_balances_at_fork_point()           // u64 micro-
units

  surviving_txs = all_txs.exclude(losing_txs)
ordered_txs = canonical_sort(surviving_txs)          // see Section 9.1

rolled_back = Set()

loop:

  newly_rolled_back = Set()

  balances = snapshot_balances_at_fork_point() // reset each iteration

  expected_nonce = {}                            // per-account
```

```
  for tx in ordered_txs:
        if tx.hash in rolled_back: continue
        // Nonce continuity: if a prior nonce was rolled back,
        // all subsequent nonces from this account are invalid
        if tx.nonce > expected_nonce[tx.from]:
          newly_rolled_back.add(tx.hash)
          continue
        cost = to_micro(tx.amount) + to_micro(tx.gas)
        if balances[tx.from] < cost:
          newly_rolled_back.add(tx.hash)
          continue
        balances[tx.from] -= cost
        balances[tx.to] += to_micro(tx.amount)
        expected_nonce[tx.from] += 1
  if newly_rolled_back.is_empty(): break // stable
     rolled_back.extend(newly_rolled_back)
  return (rolled_back, balances)
```

Convergence guarantee: Each iteration can only add rollbacks, never remove them. The valid transaction set shrinks monotonically. Since the transaction set is finite, the algorithm terminates.

Proof of convergence (termination). Let S_i be the set of surviving transactions after iteration i. The algorithm guarantees $S_{i+1} \subseteq S_i$ (monotonic shrinkage): each iteration either produces at least one newly rolled-back transaction ($|S_{i+1}| < |S_i|$) or produces no new

rollbacks (S_{i+1} = S_i, and the algorithm terminates). Since |S_i| ≥ 0 and decreases by at least 1 per non-terminal iteration, the algorithm terminates in at most |S_0| steps.

Proof of convergence (correctness). Every node computing the cascade on the same input set produces identical output because: (1) the initial losing set is determined by the deterministic conflict resolution algorithm (Section 9.4); (2) snapshot_balances_at_fork_point() returns the same u64 micro-unit balances on all nodes (anchored to a common confirmed checkpoint); (3) canonical_sort produces an identical total order on all nodes (Section 9.1); (4) the replay loop is a pure deterministic function — for each transaction in canonical order, the balance check (balances[tx.from] < cost) and nonce continuity check (tx.nonce > expected_nonce[tx.from]) produce identical boolean results given identical input state; (5) because balances are reset from the fork-point snapshot at the start of each iteration and the rolled-back set only grows, the inputs to each iteration are fully determined by the growing rolled_back set, which is itself deterministic. Therefore S_i is identical on all nodes for all i, and the final state (balances and surviving transaction set) is identical.

Complexity: Worst case O(n^2) where n is the number of transactions in the partition period. Each iteration is O(n) (replay all transactions), and at most n iterations occur. In practice, cascades are shallow — empirically, most transactions do not depend on funds from rolled-back transactions, and convergence occurs in 1–3 iterations. For a partition with k direct conflicts and an average dependency chain depth of d, the expected iteration count is O(d), which is typically O(1) for well-distributed transaction graphs.

9.6 Phase 5 - State Reconciliation

After cascade rollback completes:

1. Account state rebuild: final_balances from the cascade replay (in micro-units) are converted back to f64 and written as canonical account state.

2. DAG cleanup: Direct conflict losers (nonce reuse - malicious) are removed from the DAG entirely. Cascade victims (innocent users whose upstream funds evaporated) are marked rolled_back: true and kept for auditability.

3. Merge checkpoint: A new checkpoint is created at fork_point_height + 1 with:

- provisional: false
- previous_hash linking to the fork-point checkpoint
- finalized_tx_hashes containing all surviving transactions from both partitions
- merge_report_hash referencing the full MergeReport
- Signed by the reunified validator quorum (must meet full 2/3 total stake threshold)

4. Provisional chain retirement: All provisional checkpoints from both partitions are archived to merge_history and removed from the active checkpoint chain.

9.7 Nonce Gap Behavior

When a transaction is rolled back, all subsequent transactions from that account (with higher nonces) are also rolled back, regardless of their individual validity. This is because Rinku enforces strict nonce sequentiality - a "gap" in the nonce sequence means the account's state is undefined for all operations after the gap.

This is a deliberate design choice: it prevents subtle state inconsistencies where an account's later transactions depend on side effects (balance changes, contract state mutations) of the rolled-back transaction, even if the later transactions appear independently valid.

Why nonce-gap rollback is necessary. Consider the alternative - nonce remapping, where surviving transactions are renumbered to fill gaps. This introduces several problems: (1) transaction hashes would change (the nonce is part of the signed data), invalidating signatures; (2) any VerifiableObject referencing the original transaction hash would become invalid; (3) the remapped transaction would need re-execution for contracts, violating the determinism requirement. Partial replay (skipping the gap and continuing with later nonces) is equally problematic because later transactions may depend on state changes from the skipped transaction - a transfer at nonce 3 may only be valid because nonce 2 received incoming funds.

Nonce-gap rollback is conservative - it potentially rolls back more transactions than strictly necessary. This is an intentional safety-over-liveness tradeoff: the cost of unnecessary rollbacks (user inconvenience) is far lower than the cost of state corruption from invalid partial replays.

Important subtlety: cross-partition nonce filling. When a transaction loses a direct conflict (e.g., local nonce 1 loses to remote nonce 1), the winning transaction fills the nonce slot. Subsequent nonces (2, 3, ...) from the same account are NOT rolled back - the sequence remains unbroken. Nonce-gap cascades only occur when no transaction (from either partition) fills a nonce slot.

9.8 Transaction Classification

Rinku implements a PartitionSafety classification system that governs which transactions may execute during partition mode. This is a protocol-level enforcement, not an

application-layer convention - the gate is applied in the transaction acceptance path before DAG insertion.

Classification	AP-Mode Behavior	Transaction Types

Safe	Always allowed	DataOnly, Consolidation, Reward
BoundedSpend	Allowed within partition budget	Transfer, Contract
CpOnly	Rejected during partition	Stake, Unstake, ClaimRewards

Partition budget system. Accounts can optionally configure a partition_budget - a maximum amount spendable during any single partition epoch. When a BoundedSpend transaction is submitted during partition mode, the protocol checks:

1. Has the account configured a partition budget? If not, the transaction proceeds without restriction (but carries full rollback risk).
2. Would this transaction cause partition_budget_spent + tx_amount to exceed partition_budget? If so, the transaction is rejected.

Transactions within the partition budget are economically guaranteed to be merge-safe: even in the worst case (identical spending in both partitions), the combined spend cannot exceed the pre-partition balance if the budget is set to at most half the balance. This transforms partition tolerance from a probabilistic property to a deterministic one for opted-in accounts.

The budget is tracked via partition_budget_spent on the Account struct and is reset when the node enters a new partition epoch.

9.9 Three-Tier Receipt Model

The VerifiableObject system supports explicit receipt tiers as semantic markers:

- TentativeReceipt: Issued during partition mode. Carries provisional checkpoint anchor and partition epoch. Explicitly communicates "valid locally, subject to rollback." Applications can render these with appropriate UI affordances (e.g., a pending indicator).
- FinalReceipt: Issued after checkpoint finality during normal operation. Carries full BLS quorum signature. Irrevocable under honest majority assumption.

- ReconciliationReceipt: Issued after merge. Proves that a tentative transaction was either accepted (upgraded to final status in the merge checkpoint) or rejected (rolled back with reason code: ConflictLoser, CascadeVictim, NonceContinuityGap, InsufficientBalanceAfterConflictResolution).

These tiers are not separate data structures but semantic interpretations of the existing VerifiableObject with its provisional, partition_epoch, and merge_report_hash fields. Applications and wallets should inspect these fields to determine the receipt tier and display appropriate finality information to users.

10. Smart Contracts & WASM Runtime

10.1 Runtime Architecture

Rinku executes smart contracts in a sandboxed WASM environment built on the wasmi interpreter:

- Memory sandbox: Configurable up to 256 pages (16 MB), with bounds-checked guest memory access
- Import validation: Only rinku and env namespaces are permitted; all other imports are rejected at deployment
- Deterministic execution: wasmi provides bit-identical execution across platforms

10.2 Dual Gas Metering

Gas is metered at two levels to prevent both instruction-level abuse and host-call abuse:

- Fuel metering: Every WASM instruction consumes interpreter fuel (default budget: 10,000,000 fuel units)
- Host gas metering: Expensive host operations charge additional gas through the GasMeter

Total gas = (fuel_consumed / 100) + host_gas + base_gas + input_bytes_gas

Where input_bytes_gas = input_size * 16 (16 gas per byte of contract input).

Host operation gas schedule:

Operation Gas Cost Rationale

base_execution 1,000 Fixed cost per contract
invocation

storage_read 200 Database lookup

storage_write 5,000 Database write + state hash update

storage_delete 5,000 Database delete + state hash update

memory_alloc 3 Per-page memory allocation

transfer 8,000 Balance mutation on two accounts

mint 6,000 Token minting operation

burn 6,000 Token burning operation

emit 500 Event serialization and broadcast

hash 300 Cryptographic hash (sha256/keccak256)

balance_check 100 Ledger state lookup (get_balance/get_staked)

log 100 Debug output (no state mutation)

10.3 Host ABI

The rinku namespace exposes:

Function Description

storage_read, storage_write, storage_delete, storage_has Contract key-value storage (JSON-serialized values)

Function	Description
get_caller, get_block_height, get_timestamp, get_input	Execution context
get_contract_id	Self-referential contr
get_balance, get_staked	Ledger queries
transfer	Native token transfers

`emit_event`	Event emission for inc subscribers
`emit_view_key`	Expose state fragments verification
`sha256, keccak256`	Cryptographic hashing

10.4 Stateless dApp Standard (Proof-Carrying Contracts)

Contracts define ViewKeySpec schemas specifying which pieces of state should be exposed for external verification. Every mutating call returns a StatefulReceipt containing:

- View key values (specific state fragments selected by the contract)
- Merkle multiproof connecting those values to the checkpoint state root
- Finality certificate (checkpoint anchor with BLS signature)

This enables persistently stateless clients - applications that never store blockchain state locally but can verify any claim on demand using receipts. Concretely, a persistently stateless client works as follows: when the application needs to display data (a user's balance, a social feed, a contract's leaderboard), it queries any Rinku node for the relevant StateWitness - a sparse Merkle multiproof over the specific storage keys it needs, anchored to the latest finalized checkpoint. The node returns the witness as a VerifiableObject URL. The client verifies the Merkle proof against the checkpoint's state root and the BLS aggregate signature against the known validator set, then renders the proven data directly. It stores nothing. The next time it needs the same data, it requests a fresh witness. There is no local database, no sync process, no stale cache to invalidate. The client is stateless not just at startup but persistently - it never accumulates state that could drift from the canonical chain. The tradeoff is that the client must be able to reach at least one node to obtain fresh witnesses; it cannot serve reads while fully offline (though it can verify previously-received VOs offline indefinitely).

10.5 Contract SDK

Rinku provides a Rust SDK (rinku-contract-sdk) for contract development with the following macros and helpers:

- entrypoint! - Declares the contract's entry point function, handling input deserialization and output serialization.
- contract_init! - Declares the contract's initialization function, called once at deployment.

- contract_call! - Declares a callable contract function with automatic gas metering and error handling.

The SDK provides helper functions for common operations:

- Storage: storage::get(), storage::set(), storage::delete(), storage::has() - type-safe wrappers around the host ABI storage functions with automatic JSON serialization.
- Transfers: token::transfer(to, amount) - transfers RKU from the contract's balance.
- Events: events::emit(name, data) - emits a named event for indexing and WebSocket subscribers.
- View keys: view::expose(key, value) - registers a view key for inclusion in the StatefulReceipt.

Contracts are compiled to WASM using standard Rust toolchain targeting wasm32-unknown-unknown, then deployed via a Contract transaction with the base64-encoded WASM binary and initial state.

11. Economic Model

11.1 Supply Schedule

Parameter	Value
Maximum supply	30,000,000 RKU
Genesis allocation	6,000,000 RKU (20%)
Maximum emittable	24,000,000 RKU
Halving interval	3,150,000 checkpoints (~ intervals)
Total halvings	5
Minimum reward floor	0.122887 RKU per checkpc initial_reward / 2^5, pe not halve further)

Genesis allocation distribution. The 6,000,000 RKU genesis allocation is distributed as follows: each of the 3 genesis validators receives an initial balance sufficient to stake

GENESIS_VALIDATOR_STAKE (50,000 RKU each, totaling 150,000 RKU staked). The remaining 5,850,000 RKU constitutes the genesis reserve, held in the genesis account for future distribution via governance (not yet implemented), ecosystem grants, and faucet funding on testnet. No single entity holds a controlling share of the genesis allocation — the maximum individual holding at genesis (if all reserve were concentrated) would be ~19.5% of total supply, insufficient for unilateral quorum influence (requires >33.33%).

Halving interval derivation. The interval of 3,150,000 checkpoints is derived from the target of approximately 1 year per halving epoch at the design checkpoint interval of 10 seconds: 365.25 days × 24 hours × 3600 seconds / 10 seconds ≈ 3,155,760, rounded to 3,150,000 for a clean constant. Five halvings produce a geometric emission decay (initial reward ~3.93 RKU → 1.965 → 0.983 → 0.491 → 0.246), after which the reward floor of 0.122887 RKU applies permanently — this floor does not halve further, ensuring perpetual validator incentives. With the floor in place, total supply asymptotically approaches but never exceeds 30,000,000 RKU because the emission logic enforces min(reward, MAX_SUPPLY - current_supply).

Emission schedule:

Halving	Checkpoints	Reward per Checkpoint
0	0 – 3,149,999	~3.93 RKU
1	3,150,000 – 6,299,999	~1.965 RKU
2	6,300,000 – 9,449,999	~0.983 RKU
3	9,450,000 – 12,599,999	~0.491 RKU
4	12,600,000 – 15,749,999	~0.246 RKU
5+	15,750,000+	0.122887 RKU (floor)

The emission curve follows a geometric decay with a floor, ensuring that block rewards never reach zero - validators always have a base-layer incentive to participate, even after the majority

of tokens have been emitted. The hard cap of 30M RKU is asymptotically approached but never exceeded due to the floor mechanism and total supply enforcement in the emission logic.

11.2 Checkpoint Rewards

Checkpoint rewards are distributed when a checkpoint is finalized. The reward amount is determined by the emission schedule (Section 11.1) and is distributed to all active validators proportional to their effective weight:

Distribution formula:

validator_reward = checkpoint_reward * (validator_effective_weight / total_effective_weight)

Where effective_weight is computed using the dual-weight system (Section 11.3). The total checkpoint reward is minted as new supply, subject to the hard cap - if minting the full reward would exceed 30,000,000 RKU, the reward is reduced to the remaining mintable amount.

Rewards are credited directly to the validator's liquid balance (not to their staked balance), allowing validators to compound their stake through explicit re-staking or use rewards for other purposes.

11.3 Weighted Proof-of-Stake (WPoS) Reward Distribution

Checkpoint rewards are distributed through a dual-weight system. Note: the "weight" in WPoS reward distribution is distinct from the DAG transaction weight described in Section 7.3. DAG weight determines transaction ordering priority and conflict resolution; WPoS weight determines validator reward shares.

- Stake weight (70%): Proportional to amount staked. The 70/30 split is calibrated to ensure that stake remains the dominant factor in reward distribution (preventing age-only validators from receiving outsized rewards) while providing a meaningful incentive for continuous participation.
- Age weight (30%): Rewards long-term active participation; requires minimum 100 RKU bond; decays 10% per missed checkpoint. Age weight is computed as min(checkpoints_since_stake / TARGET_AGE, 1.0) where TARGET_AGE is 1000 checkpoints (~2.7 hours). This ramps linearly from 0 to 1.0 over the target period, rewarding validators who maintain continuous uptime. The 10% decay per missed checkpoint penalizes intermittent validators.

11.4 Staking Requirements

Parameter	Value
Minimum stake	100 RKU
Minimum stake age for rewards	15 seconds
Unstake cooldown	24 hours
Slashing unbonding period	14 days

11.5 Additional Reward Streams

- Tip rewards (1%): Distributed as incentives for specific network actions
- Witness rewards (0.2%): Incentivize DAG connectivity by rewarding nodes that reference other transactions as parents

11.6 Gas Fee Model

Rinku implements an EIP-1559-inspired dynamic gas pricing mechanism:

Parameter	Value
Target throughput	150 transactions per 15-
Adjustment factor	12.5% per period
Minimum gas price	0.001 RKU
Maximum gas price	10.0 RKU

The gas price adjusts each period: if the actual transaction count exceeds the target, the price increases by the adjustment factor; if below target, it decreases. This creates a self-regulating fee market that responds to demand without requiring explicit fee auctions.

11.7 Adaptive Fee Burn

Transaction fees are split between burning and validator rewards:

- Burn ceiling: 30% of fees
- Burn scaling: Burn percentage increases linearly as circulating supply approaches 50% of the hard cap. At 0% of max supply, no fees are burned; at 50% (15M RKU), the full

30% ceiling is reached.
- Validator floor: Validators always receive at least 70% of fees

The adaptive burn creates deflationary pressure that increases as the token supply grows, counterbalancing emission inflation and creating a natural equilibrium. At maturity (when emission is at the floor rate), the burn mechanism may offset or exceed emission, potentially making the token supply effectively stable or mildly deflationary depending on transaction volume.

11.8 Micro-Unit Precision

All internal accounting uses u64 micro-units with 8 decimal places (1 RKU = 100,000,000 micro-RKU). This eliminates floating-point precision errors, which is particularly critical for deterministic merge reconciliation (Section 9).

Boundary disclosure. The AccountStateProof API response struct intentionally carries both canonical u64 micro-unit fields (balance_micro, staked_micro) and f64 display fields (balance, staked) for JSON readability. The f64 fields are derived from micro-units via from_micro_units() at serialization time and are provided for human consumption only. All consensus-critical operations — merge reconciliation, balance checks, gas enforcement, reward calculation — operate exclusively on u64 micro-units. Cross-network state exchange (sync snapshots, gossip messages) also uses u64. The f64 representation is lossy for values exceeding 2^53 micro-units (~90,071,992 RKU), which is above the hard cap and therefore does not affect correctness in practice.

12. Slashing & Economic Security

12.1 Violation Types

Violation	Severity	Penalty
Nonce Reuse (Cross-Partition Double-Spend)	Malicious	10% bal stake s reputat
Cross-Partition Economic Overdraft	Gray area	0.10 re linear checkpc
Cascade Victim	Innocent	No pena

Double-Sign (same height, different hash)	Malicious	15% sta
Invalid Checkpoint Proposal	Malicious	25% sta
Receipt/Proof Tampering	Malicious	25% sta
Invalid Proof/Witness Submission	Malicious	15–20%
Liveness Failure (3+ missed checkpoints)	Negligent	5% stak 10% for

The graduated penalty structure reflects a key design principle: the protocol distinguishes intent. Nonce reuse requires deliberate action and is treated as malicious. Economic overdraft may be accidental and receives a soft, recoverable penalty. Cascade victims are blameless and bear no cost.

12.2 Reputation & Weight Modifier

Accounts with reputation penalties receive reduced weight in all weight calculations:

effective_weight = base_weight * (1.0 - reputation_penalty)

This reduces the influence of penalized accounts in DAG weight calculations, tip selection, and consensus voting without requiring immediate ejection from the network.

12.3 Penalty Decay

Cross-partition overdraft reputation penalties decay linearly over 100 checkpoints (~16 minutes at 10s intervals), allowing honest users caught in an ambiguous situation to recover their standing. Nonce-reuse penalties are permanent and do not decay.

The decay is computed as:

remaining_penalty = original_penalty * max(0, 1.0 - (current_height - penalty_height) / 100)

Where penalty_height is the checkpoint at which the penalty was applied (stored in penalty_decay_checkpoint on the Account struct).

12.4 Game-Theoretic Analysis

Nonce-reuse attack cost. An attacker attempting a cross-partition double-spend faces: 10% balance confiscation + 100% stake slash + 0.50 permanent reputation penalty. For this to be profitable, the double-spent amount must exceed 0.10 * balance + 1.00 * stake + NPV(weight_reduction). The permanent reputation penalty means the attacker's future transaction weight is halved indefinitely, reducing their influence in all protocol interactions. For any account with meaningful stake, the expected cost far exceeds the maximum double-spend gain (which is bounded by the account's pre-partition balance).

Economic overdraft opportunism. An attacker who intentionally spends in both partitions faces a 0.10 reputation penalty with 100-checkpoint decay. The cost is temporary weight reduction. This penalty is intentionally mild because the attack surface is limited (the attacker can only spend their own balance) and the behavior may be innocent. The partition budget system provides a stronger mitigation for users who want guaranteed merge safety.

Cascade attack analysis. An attacker cannot deliberately cause cascades against specific victims without first losing their own funds in a direct conflict. Cascade rollbacks are a second-order effect - they require the attacker to sacrifice their own transaction first. The attacker bears the full penalty for the initiating conflict; cascade victims bear no penalty. The attacker cannot profit from cascading because the rolled-back funds return to their pre-partition state, not to the attacker.

12.5 Acknowledged Security Considerations

The following security properties are areas of active analysis and represent known open questions in the current protocol design:

Nothing-at-stake for fast-path ACKs. Validators have no explicit penalty for selective ACK withholding — a validator can delay or omit ACKs for competitor transactions to slow their

fast-path certification. Mitigation: fast-path is the primary finality mechanism for transfers, so withheld ACKs can delay transaction finality. However, liveness failure penalties (Section 12.1) provide indirect deterrence for persistent non-participation, and the 2/3 quorum threshold means any individual validator's withholding is insufficient to prevent certification as long as the remaining stake exceeds the threshold. Formal analysis of selective withholding incentives is future work.

Partition budget replay via rapid cycling. The partition budget resets per partition epoch. An adversary who can rapidly induce partition cycling (NORMAL → PARTITIONED → NORMAL → PARTITIONED) could multiply the effective budget by the number of cycles. Mitigation: the T_conf (30s) and T_recovery (10s) timeouts impose a minimum 40-second cycle time, bounding the cycling rate. Additionally, rapid cycling itself requires control over network connectivity affecting >1/3 of stake, which is a strong adversary assumption. A minimum inter-epoch delay or cumulative budget across epochs is under consideration.

Merge strategic delay. A validator whose transactions will lose conflict resolution has an incentive to delay broadcasting their MergePayload. Currently, no explicit timeout enforcement or penalty for delayed merge participation is described. In practice, merge is orchestrated by the reconnecting nodes and does not require the cooperation of the validator whose transactions lose — the winning partition's state is adopted. Formal timeout-based merge liveness guarantees are future work.

Slashing evidence authentication. SlashingEvidence gossip messages carry the conflicting signed messages as proof. The evidence itself is self-authenticating — the conflicting signatures are verifiable by any node against the validator's public key. However, the gossip message wrapper is not signed by the evidence submitter, meaning any node can relay (but not forge) slashing evidence. This is by design: evidence should be freely relayable, and forgery is impossible because the evidence contains the validator's own signatures.

VO proof replay within freshness window. The max_age_checkpoints mechanism prevents stale proof replay but does not prevent repeated submission of the same valid proof within the freshness window. For BYOP contract interactions, this means a contract must implement its own replay protection (e.g., tracking consumed proof hashes) if single-use semantics are required. The protocol provides the freshness primitive; application-layer replay protection is the contract's responsibility.

BYOP front-running. The receipt-composition pattern (Section 3.4) is vulnerable to front-running: an attacker observing a pending BYOP transaction can submit a competing transaction with a more favorable oracle receipt within the freshness window. This is a general MEV concern shared with all receipt-composable systems. Mitigation strategies include shorter freshness windows, commit-reveal schemes at the contract layer, and encrypted mempool proposals (future work).

Signer bitmap authentication. The signer bitmap identifying which validators signed a checkpoint is not independently signed — it is embedded within the checkpoint data structure alongside the aggregate BLS signature. The bitmap's integrity is implicitly verified: if the bitmap is tampered with (claiming additional signers), the aggregate public key reconstruction will not match the aggregate signature, and verification will fail. An attacker cannot inflate the apparent quorum without possessing the corresponding BLS private keys.

Validator set evolution for offline VO verification. A VO generated at checkpoint N cannot be verified offline by a party holding only the genesis validator set if the validator set changed between genesis and checkpoint N. Offline verification requires the verifier to hold a validator set that was active at the VO's checkpoint height. For long-lived offline verification, VOs should embed the signer membership proofs (Section 4.2, item 5) that chain back to a known anchor. Full validator set evolution proofs (chaining validator set changes through signed checkpoints) are future work.

Validator exit before slashing. A validator that double-signs could attempt to front-run slashing by submitting an unstake transaction during the 24-hour cooldown. Mitigation: the 14-day unbonding period (Section 11.4) means staked tokens remain slashable for 14 days after unstaking, regardless of when the unstake was initiated. Slashing evidence submitted during the unbonding period is applied to the locked stake. A double-signer cannot escape slashing by unstaking.

Sub-linear stake and Sybil resistance. The sub-linear stake weight formula (stake^0.5 * 2.0) means that splitting stake across N identities yields higher aggregate weight than a single large stake (N validators at stake S/N each have total weight N * (S/N)^0.5 * 2.0 = 2.0

- S^0.5 * N^0.5). Sybil resistance is therefore load-bearing on the minimum stake requirement (100 RKU): creating N Sybil validators requires N * 100 RKU, limiting the splitting advantage. The practical break-even point where splitting becomes profitable depends on the attacker's total stake relative to the minimum.

13. Networking & P2P Protocol

13.1 Gossipsub

Rinku's gossip protocol operates on the rinku/1.0.0 topic with the following message types:

Message Type	Priority	Descr
Transaction	Normal	Signe propa

Message Type Priority Description

TipAnnouncement Normal Current DAG tips and size (triggers sync if peer is ahead)

CheckpointAnnouncement High New finalized checkpoint with transaction bodies

CheckpointSignature High Individual validator signature for checkpoint voting

TxConfirmBroadcast High Fast-path broadcast for per-transaction certification

TxConfirmAck High Validator acknowledgment for fast-path voting

BloomAnnouncement Low Bandwidth-efficient advertisement of known transactions

PeerDiscovery Low Shares known peer addresses for mesh expansion

ConflictResolution Normal Broadcasts conflict resolution decisions

SlashingEvidence High Proof of validator misbehavior (double-signing)

WeightVote Normal Validator vote for transaction trust weighting

MergePayload High Partition merge request with DAG delta

MergeResult High Partition merge response with reconciliation data

SyncRequest Normal Request for missing transactions

Message Type	Priority	Desc
SyncResponse	Normal	Resp tran

Bloom filter optimization. To reduce bandwidth, nodes periodically broadcast BloomAnnouncement messages containing a Bloom filter of their known transaction hashes. The filter uses double SHA-256 hashing for optimal bit distribution. Before sending a transaction to a peer, the node checks the peer's Bloom filter - if the transaction is likely already known, it is not retransmitted. The filter includes the node's checkpoint height and tip count, enabling peers to detect if they are behind and should initiate sync.

Propagation batching. To prevent message storms under high transaction load, the gossip service uses a background propagation task with a MAX_PROPAGATION_BATCH of 100 transactions. Pending transactions are accumulated and broadcast in batches, amortizing the per-message overhead of gossipsub.

Deduplication. A BoundedHashSet (known_txs) tracks recently seen transaction hashes to prevent infinite gossip loops. The set has a bounded capacity and evicts oldest entries when full.

13.2 Lock-Free Message Handling

Rinku's P2P receive path is designed to eliminate mutex contention on the critical message-processing path:

Channel-based architecture. The NetworkHandle wraps the libp2p swarm and exposes message channels. During initialization, the gossip message receiver (message_rx) and sync request receiver (sync_incoming_rx) are extracted from the NetworkHandle before it is wrapped in Arc<Mutex<>>. These channels are passed directly to handler tasks via set_p2p_channels().

Implications. The message receive path never acquires the NetworkHandle mutex. Incoming gossip messages flow from the libp2p event loop → mpsc channel → run_p2p_receiver task → handle_message processing, entirely lock-free. Similarly, sync requests flow through a separate channel to run_sync_request_handler. Sync response sending uses a cloned command_tx sender, which is also lock-free (mpsc senders are cheaply cloneable).

This architecture eliminates the 5-25ms polling latency per message that would occur if the receive path needed to acquire a mutex on every incoming message, which is critical for maintaining sub-second fast-path finality latency under load.

13.3 Connection Management

Idle timeout. Connections have a 600-second idle timeout, configured to prevent premature KeepAliveTimeout disconnects on low-traffic deployments where minutes may pass between gossip messages.

Mesh maintenance. The network service periodically checks if the number of validated peers is below MIN_MESH_PEERS. If the mesh is unhealthy, it re-dials bootstrap peers to restore connectivity. The InsufficientPeers publish error (which occurs during startup or reconnection when no gossipsub peers are available) is logged at trace level to avoid log noise during expected transient states. Note: MIN_MESH_PEERS is currently set to 1, which is a testnet configuration suitable for small validator sets (3 nodes). In production, this value must be increased to prevent a single Sybil peer from capturing a validator's entire network view. A production recommendation of MIN_MESH_PEERS ≥ 3 (or a fraction of the validator set) will be established based on mainnet validator set size.

Peer discovery. Nodes exchange PeerDiscovery messages containing their known peer addresses. New peers are added to the connection pool and validated against the validator

identity service. The /api/peers endpoint exposes the current P2P peer list as the primary field, with legacy HTTP peer information included only when non-empty.

Validator identity verification. The ValidatorIdentityService maps P2P peer IDs to validator addresses and BLS public keys. When a new peer connects, the service verifies the peer's claimed identity against the registered validator set. This prevents Sybil attacks at the network layer - only peers corresponding to staked validators contribute to stake visibility calculations for partition detection.

14. Related Work

Rinku draws on and extends several lines of prior work in distributed ledgers, partition-tolerant systems, and proof-carrying architectures. This section positions Rinku honestly within the landscape, crediting the systems that inspired specific design choices and identifying where Rinku's combination of properties is genuinely novel.

14.1 DAG-Based Ledgers

DAG-based transaction structures have been explored extensively as alternatives to linear blockchains:

IOTA (Tangle). IOTA [6] pioneered the use of a DAG for transaction ordering in a distributed ledger, with each transaction approving two prior transactions. The original Tangle design targeted IoT microtransactions and eliminated mining fees. However, IOTA relied on a centralized Coordinator for finality until its Coordicide effort, and the Tangle's tip selection algorithm was vulnerable to parasite chain attacks. IOTA does not provide partition tolerance - the Coordinator is a single point of failure for finality. Rinku shares the DAG structure but differs fundamentally in its consensus mechanism (dual-lane BLS-certified fast-path finality with QCC checkpoints) and its explicit partition-tolerant design.

Nano (Block Lattice). Nano [7] uses a block-lattice structure where each account maintains its own chain. This enables asynchronous, feeless transactions with near-instant confirmation via delegated proof-of-stake voting. Nano's design is elegant for simple value transfers but does not support smart contracts or programmable state. Nano does not address network partitions - its Open Representative Voting assumes persistent connectivity to delegates. Rinku's account-nonce model is structurally similar to Nano's per-account chains but embeds transactions in a shared DAG with explicit cross-account ordering via checkpoints.

Sui (Mysticeti). Sui [3] developed the Mysticeti consensus protocol, which achieves sub-round-trip latency by using an uncertified DAG where blocks themselves serve as implicit votes. Rinku's RCC architecture shares the intuition that simple transfers can be finalized faster than shared-state operations, but implements this via a dual-lane split: Fast Lane transactions

(transfers, staking) achieve sub-second BLS-certified finality through explicit TxConfirmAck voting, while Contract Lane transactions are ordered through QCC checkpoints. Sui achieves a similar separation via its owned-object fast path versus consensus-ordered shared objects. Sui does not address partition tolerance — Mysticeti halts when the network cannot reach quorum. See Section 6 for the full RCC architecture.

Avalanche. Avalanche [9] introduced the concept of metastable consensus through repeated sub-sampled polling, achieving probabilistic finality without leaders. RCC draws on this insight but differs in two key ways: (1) RCC uses a single broadcast round where all validators participate (rather than repeated random sub-sampling), achieving deterministic 2/3-stake fast-path certification in a single network round-trip; (2) RCC restricts fast-path finality to single-owner operations, routing shared-state contract calls through checkpoint ordering. Avalanche does not provide partition tolerance with deterministic reconciliation.

Other DAG protocols. Hashgraph [8] uses a gossip-about-gossip protocol with virtual voting; Bullshark and Tusk use certified DAGs with structured commit rules. A comprehensive survey of DAG-based consensus is provided by Raikwar et al. [2]. Among these, none provide explicit partition tolerance with deterministic reconciliation - all assume or require persistent quorum reachability for safety and liveness.

14.2 Partition-Tolerant Distributed Systems

SwarmDAG. SwarmDAG [1] is the direct inspiration for Rinku's partition tolerance design. Tran et al. proposed a partition-tolerant distributed ledger for swarm robotics using a DAG structure that continues operating during network fragmentation. SwarmDAG demonstrated the viability of DAG-based ledgers in intermittently connected environments and introduced the concept of merge-on-reconnect for divergent DAG state. Rinku extends SwarmDAG's approach in several directions: a formalized 5-phase merge protocol with provable deterministic convergence (Section 9), a transaction classification system that governs which operations are safe during partitions (Section 9.8), graduated economic penalties that distinguish intentional abuse from accidental conflicts (Section 12), and a partition budget system that provides deterministic merge-safety guarantees for opted-in accounts.

CRDTs and eventual consistency. Conflict-free Replicated Data Types (CRDTs) guarantee convergence without coordination by restricting operations to commutative, associative, and idempotent structures. Rinku does not use CRDTs for its core financial state because token transfers are inherently non-commutative - order matters and conflicts have economic consequences. However, CRDTs are identified as a natural extension for contract storage types (Section 15.1) where merge-friendly data structures would be safe for unrestricted partition-mode operation.

Bayou and anti-entropy. Bayou [10] pioneered application-level conflict resolution with dependency tracking and tentative/committed state separation. Rinku's three-tier receipt model

(tentative, final, reconciliation; Section 9.9) follows a similar conceptual structure, and the distinction between provisional and finalized state maps directly to Bayou's tentative/committed semantics.

14.3 Proof-Carrying and Lightweight Client Approaches

SPV (Simplified Payment Verification). Bitcoin's SPV model [4] proves transaction inclusion via Merkle paths but requires the verifier to trust block headers, which in turn requires following the longest-chain rule. SPV proofs do not prove account state, only transaction inclusion, and the verifier must maintain a header chain or trust a provider. Rinku's self-contained proofs carry the full verification context (Section 4.2), eliminating the need for any ongoing chain-following.

IBC Light Clients. The Inter-Blockchain Communication protocol uses light clients that track validator committees and their signatures. Light clients require an ongoing connection to at least one honest full node to follow validator set evolution. Without this connection, the client's view becomes stale and unverifiable. Rinku's VerifiableObjects are self-contained at the cost of embedding more data per proof (~2KB for an account proof), but eliminate the ongoing connection requirement entirely.

Stateless Ethereum proposals. Ethereum's research into statelessness (Verkle trees, witness-based block execution) aims to reduce full-node storage requirements by embedding state access proofs in blocks. This is a block-producer optimization - clients still need to follow the chain. Rinku's proof-carrying contracts (Section 10.4) target a different use case: enabling application clients that never follow the chain at all, instead verifying individual claims on demand.

14.4 Differentiation

No single existing system combines all five of the properties that Rinku targets. The following table compares across the primary axes:

Property	IOTA	Nano	Sui/Mysticeti	SwarmD
DAG-based transaction structure	Yes	Yes (block-lattice)	Yes (uncertified DAG)	Yes
Sub-second transaction finality	No (Coordinator)	Yes (~0.2s)	Yes (~0.4s)	No

Deterministic partition tolerance	No	No	No (halts)	Yes merg
Self-contained offline proofs	No	No	No	No
Programmable smart contracts	Yes (ISC)	No	Yes (Move)	No

The individual columns are not novel - DAG structure, fast finality, partition tolerance, proof portability, and smart contracts all exist independently in the literature. Rinku's contribution is the specific combination: a system that provides all five simultaneously, with the partition tolerance and proof portability designed to reinforce each other (VerifiableObjects remain verifiable during and after partitions because they carry their own verification context).

This combination is motivated by the target deployment environments described in Section 1: mesh networks and intermittently connected systems where both partition tolerance and infrastructure-free verification are necessary, not optional.

15. Future Work

15.1 CRDT-Compatible State Types

For contract storage, introduce merge-friendly data types (sets, append-only logs, max counters, OR-maps) that can be safely updated during partitions without conflict. Ideal for social, messaging, and collaborative applications. Would integrate with the transaction classification system (Section 9.8) to automatically determine AP-safety - contracts that exclusively use CRDT-compatible state types could be classified as Safe rather than BoundedSpend.

15.2 Object Ownership Model

Explore single-writer object ownership where owned objects can be processed during partitions without conflict, while shared objects remain CP-only. This model is inspired by Sui's object-centric programming model: if an object has a single owner, mutations by that owner are inherently conflict-free across partitions. The challenge is integrating this with Rinku's account-based (rather than object-based) state model.

15.3 Cross-Chain Proof Composability

Leverage BYOP and VerifiableObjects for cross-chain interoperability. Since VOs are self-contained and carry their own verification data, a Rinku VO could be submitted to a contract on another chain (or vice versa) as a ProofInput. The receiving chain would verify the proof's BLS signature against a registered Rinku validator set root. This pattern enables receipt-composable bridges without trusted relayers - the bridge contract verifies the proof's mathematical validity rather than trusting a third party to relay state.

15.4 Contract-Level Merge Hooks

Allow contracts to define custom merge resolution logic for their storage, replacing the default "last-write-wins by weight" rule with application-aware conflict resolution. This would address the pathological contract state corruption risk identified in Section 9.1. A contract could implement a merge_resolve(key, local_value, remote_value, local_weight, remote_weight) -> value function that is invoked during Phase 5 of the merge protocol for each conflicting storage key.

15.5 Zero-Knowledge Privacy Layer

An optional ZK-SNARK layer for privacy-preserving transactions is under investigation. The goal is to enable sender obfuscation and private value transfers using Groth16 proofs, with verification integrated into the WASM contract runtime. Key design questions remain open: how

ZK proofs interact with the partition tolerance model (provisional proofs cannot be verified without the proving key, which may not be available in disconnected environments), the tradeoff between proof generation latency and transaction confirmation speed, and whether ZK privacy should be a protocol-level feature or a contract-layer concern. Initial prototyping with circomlib circuits has been completed; production integration is pending resolution of these design questions.

16. Conclusion

Rinku occupies a distinct position in the distributed ledger design space. Rather than making a fixed CAP tradeoff, it dynamically navigates the consistency-availability spectrum based on network conditions: sub-second fast-path certified finality for transfers when the network is connected, checkpoint-ordered execution for contracts, provisional availability during partitions, and deterministic convergence when partitions heal.

The system is complemented by several infrastructure layers that improve its production readiness: write-set conflict detection (Section 6.7) enables non-conflicting contract calls to achieve fast-path finality alongside transfers, micro-checkpoints (Section 6.8) provide sub-second proof availability via 200ms SMT snapshots, a write-ahead log (Section 6.10)

ensures crash recovery consistency, and a data availability layer (Section 6.11) separates consensus from data distribution via Reed-Solomon erasure coding.

The core contribution is not the mode-switching mechanism itself - dynamically adjusting consistency levels is a well-studied concept in distributed systems. The contribution is the set of protocol mechanisms that make this approach practical for a financial ledger:

1. The transaction classification system (Section 9.8) transforms partition tolerance from a binary property (available or not) into a graduated spectrum. Safe operations continue without restriction; bounded-spend operations proceed within configurable risk limits; consensus-critical operations halt until quorum is restored. This gives applications and users explicit control over their partition-mode risk exposure.

2. The 5-phase merge protocol (Section 9) provides deterministic reconciliation with formally provable convergence. Integer micro-unit accounting eliminates floating-point nondeterminism. Canonical transaction ordering ensures all nodes compute identical results. The cascade rollback algorithm traces economic dependencies exhaustively, preventing subtle state corruption from partial replays.

3. Graduated economic deterrence (Section 12) distinguishes intent. Deliberate double-spending (nonce reuse) is expensive and permanent. Accidental overdrafts receive soft, recoverable penalties. Cascade victims bear no cost. This penalty structure makes rational exploitation unprofitable while avoiding punishing honest users caught in ambiguous situations.

4. VerifiableObjects (Section 3) collapse the infrastructure requirements for trust. A rinku://vo/ URL carries everything needed to verify a claim - no full node, no RPC endpoint, no ongoing network connection. This makes Rinku's proofs inherently portable, shareable, and composable - they can be passed as transaction parameters (BYOP), embedded in QR codes, or verified entirely offline.

5. Proof-carrying contracts (Section 10.4) extend this portability to smart contract state. Contracts define which state fragments should be provable; every execution produces a StatefulReceipt with Merkle proofs and finality certificates. Applications can be persistently stateless - rendering verified data from receipts rather than maintaining local state.

Together, these mechanisms create a distributed ledger designed for environments where network partitions are a routine operating condition rather than an exceptional failure. Rinku does not claim to solve the fundamental impossibility results of distributed systems - it claims to make a practically useful navigation of those constraints for the specific domain of decentralized financial infrastructure in mesh-native environments.

Appendices

A. Formal Definitions

Definition 1 (Safety). The Rinku protocol satisfies safety if no two finalized (non-provisional) checkpoints at the same height contain conflicting state roots. Under the honest majority assumption (>2/3 stake honest), safety holds because: (a) checkpoint BLS aggregate signatures are verified at reception against the known validator set with a 2/3 quorum requirement (Section 6.3), so accepting a conflicting checkpoint requires >1/3 stake to equivocate (sign two different hashes at the same height); and (b) double-signing is detected by the ConsensusService and triggers a 15% stake slash. Note: checkpoints received during snapshot sync of legacy data may lack BLS signatures and are accepted on the basis of prev_hash chain linkage only; full BLS enforcement applies to all real-time checkpoint announcements.

Definition 2 (Liveness - Normal Mode). The protocol satisfies liveness in normal mode if every submitted valid transaction is eventually included in a finalized checkpoint, assuming >2/3 stake is reachable and the leader election mechanism produces a live leader within bounded time. Fast Lane transactions additionally achieve sub-second finality through fast-path certification before checkpoint inclusion. The leader fallback mechanism (Section 6.5) ensures liveness even if the primary leader is offline.

Definition 3 (Liveness - Partition Mode). The protocol satisfies partition liveness if every submitted valid transaction classified as Safe or BoundedSpend (within budget) is included in a provisional checkpoint within bounded time, assuming the local partition contains ≥1/3 total stake. CpOnly transactions do not satisfy liveness during partition by design.

Definition 4 (Convergence). The merge protocol satisfies convergence if, given the same set of transactions from both partitions and the same fork-point state, every node independently computes identical post-merge state. This follows from: (a) integer micro-unit accounting eliminating floating-point nondeterminism, (b) canonical ordering producing a strict total order, and (c) the cascade rollback algorithm being a deterministic function of its inputs with provable termination.

Definition 5 (Cascade Termination). Let S_0 be the initial set of surviving transactions (after conflict resolution). The cascade rollback algorithm produces a sequence $S_0 \supseteq S_1 \supseteq S_2 \supseteq \ldots$ that converges in at most $|S_0|$ iterations. Proof: each iteration either removes at least one transaction ($|S_{i+1}| < |S_i|$) or produces no new rollbacks ($S_{i+1} = S_i$, and the algorithm terminates). Since $|S_i| \geq 0$ and decreases by at least 1 per non-terminal iteration, the algorithm terminates in at most $|S_0|$ steps.

B. Parameter Reference

Parameter	Value	Section
MAX_SUPPLY	30,000,000 RKU	11.1
GENESIS_ALLOCATION	6,000,000 RKU	11.1
HALVING_INTERVAL	3,150,000 checkpoints	11.1
MIN_REWARD_FLOOR	0.122887 RKU	11.1
CHECKPOINT_INTERVAL	~10 seconds	6.2
FAST_PATH_QUORUM	2/3 total validator stake (≈66.67%)	6.1
QCC_QUORUM	2/3 total stake (≈66.67%)	6.3

Parameter Value Section Description

MAX_FAST_PATH_POOL 5,000 entries 6.1 Fast-path tracking map cap (cleared per checkpoint)

PARALLEL_QCC_DEADLINE_MS 4,000 ms 6.3 QCC vote collection deadline

SUPER_MAJORITY 75% total stake 6.3 Higher-security operations

LEADER_TIMEOUT 45 seconds 6.5 Fallback leader election trigger

PROPAGATION_GRACE 5,000 ms 6.3 Transaction propagation window

MIN_STAKE 100 RKU 11.4 Minimum validator stake

UNSTAKE_COOLDO 24 hours 11.4 Stake withdrawal
WN delay

UNBONDING_PERI 14 days 11.4 Slashing vulnerability
OD window

MIN_GAS_PRICE 0.001 RKU 11.6 Gas price floor

MAX_GAS_PRICE 10.0 RKU 11.6 Gas price ceiling

GAS_TARGET 150 tx / 15s 11.6 Target transaction
throughput

GAS_ADJUSTMENT 12.5% 11.6 Per-period price
adjustment

BURN_CEILING 30% 11.7 Maximum fee burn
percentage

MAX_SAMPLED_TIP 16 7.2 Maximum parent
S references per tx

PARTITION_THRES 66.66% visible stake 8.1 Partition detection
HOLD threshold

Parameter Value Section Description

T_CONF 30 seconds 8.1 Partition confirmation
timeout

T_RECOVERY 10 seconds 8.1 Quorum recovery
window

PROVISIONAL_FLO 33.33% total stake 8.3 Minimum stake for
OR provisional
checkpoints

WEIGHT_PROXIMIT 1.5x 9.4 Weight tiebreaker
Y proximity threshold

NONCE_REUSE_BA 10% 12.1 Balance confiscation
LANCE_PENALTY for double-spend

NONCE_REUSE_ST 100% 12.1 Stake slash for
AKE_SLASH double-spend

NONCE_REUSE_RE 0.50 permanent 12.1 Permanent reputation
PUTATION penalty

OVERDRAFT_REPU 0.10 decaying 12.1 Recoverable
TATION reputation penalty

REPUTATION_DECA 100 checkpoints 12.3 Linear decay window
Y_PERIOD (~16 min)

DOUBLE_SIGN_SLA 15% 12.1 Stake slash for
SH checkpoint
equivocation

IDLE_TIMEOUT 600 seconds 13.3 P2P connection idle
timeout

MIN_MESH_PEERS 1 (testnet; production 13.3 Minimum gossipsub

TBD) mesh size

MAX_PROPAGATIO 100 13.1 Transaction
N_BATCH propagation batch
size

Parameter	Value	Section
WASM_MAX_PAGES	256 (16 MB)	10.1
WASM_DEFAULT_FUEL	10,000,000	10.2
MICRO_UNIT_SCALE	10^8	11.8
MICRO_CP_INTERVAL	200 ms	6.8
MAX_MICRO_CHECKPOINTS	64	6.8

C. Benchmarks (Preliminary Testnet Data)

The following benchmarks were collected on a 3-validator Fly.io testnet deployment and represent preliminary testnet performance. Production benchmarks with larger validator sets and realistic network conditions are pending.

C.1 Throughput

TODO

C.2 Fast-Path Finality Latency

Percentile	Latency
p50	43–44 ms
p95	~200 ms
p99	~500 ms
Min	~22 ms

Measured as time from submission to fast-path certificate formation. Methodology note: only confirmed samples are included; transactions that did not achieve fast-path certification within 10s are excluded. The exclusion rate will be reported in production benchmarks. The p50 of ~43ms demonstrates sub-second finality for the majority of transactions via fast-path certified consensus.

C.3 QCC Checkpoint Latency

Percentile	Latency
p50	~10,300 ms
p95	~10,400 ms
p99	~15,400 ms

Checkpoint latency measures the time from transaction submission to inclusion in a Quorum-Certified Checkpoint. This is dominated by the checkpoint interval (~5-10s). Note that Fast Lane transactions achieve finality much earlier via fast-path certification (Section 6.1, Appendix C.2); QCC checkpoint inclusion provides the anchoring point for proof generation, DAG pruning, and Contract Lane execution ordering.

C.4 Proof Generation & Size

Proof Type	Generation Time	Response Size
Account proof (Merkle inclusion)	21 ms	1,953 B
Transaction proof	26 ms	1,841 B
Self-contained proof (VO URL)	21 ms	1,703 B
Batch proof (multi-receipt)	22 ms	In development

All individual proof types are generated in under 30ms. Self-contained VerifiableObject URLs encode at ~1.7KB, enabling URL-portable verification without external state. Account proofs include the Sparse Merkle Trie inclusion path and weigh ~2KB. BatchProof aggregation (shared checkpoint context and Merkle multiproof across multiple receipts) is implemented at the type level but end-to-end benchmarks including response size and multi-receipt verification are pending completion.

C.5 Fast-Path Proof (Micro-Checkpoint) Availability

Metric	Value
Proof availability after fast-path confirmation	~200ms (next micro-che
Micro-checkpoint generation interval	200ms
Proof generation time (from micro-checkpoint SMT)	<30ms
Total time from TX submission to proof available	~250-500ms (fast-path micro-checkpoint)

Fast-path proofs are available nearly sub-second after transaction submission, compared to ~10s for QCC checkpoint proofs. The micro-checkpoint ring buffer (64 entries, ~12.8s window) ensures proofs remain available until the next QCC checkpoint supersedes them.

References

[1] Tran, J.A., Ramachandran, G.S., Shah, P.M., Danilov, C., Santiago, R.A., Krishnamachari, B. "SwarmDAG: A Partition Tolerant Distributed Ledger Protocol for Swarm Robotics." Ledger, Vol. 4, Supplement 1, pp. 25–31, 2019. DOI: 10.5195/ledger.2019.174

[2] Raikwar, M., Polyanskii, N., Müller, S. "SoK: DAG-based Consensus Protocols." arXiv:2411.10026, 2024. https://arxiv.org/abs/2411.10026

[3] Babel, K., Chursin, A., Danezis, G., Kichidis, A., Kokoris-Kogias, L., Koshy, A., Sonnino, A., Tian, M. "Mysticeti: Reaching the Limits of Latency with Uncertified DAGs." arXiv:2310.14821, 2023. https://arxiv.org/abs/2310.14821

[4] Nakamoto, S. "Bitcoin: A Peer-to-Peer Electronic Cash System." 2008. https://bitcoin.org/bitcoin.pdf

[5] Boneh, D., Drijvers, M., Neven, G. "BLS Multi-Signatures With Public-Key Aggregation." 2018. https://crypto.stanford.edu/~dabo/pubs/papers/BLSmultisig.html

[6] Popov, S. "The Tangle." IOTA Foundation, 2018. https://assets.ctfassets.net/r1dr6vzfxhev/2t4uxvslqk0EUau6g2sw0g/45eae33637ca92f85dd9f4a3a218e1ec/iota1_4_3.pdf

[7] LeMahieu, C. "Nano: A Feeless Distributed Cryptocurrency Network." 2018. https://nano.org/en/whitepaper

[8] Baird, L. "The Swirlds Hashgraph Consensus Algorithm: Fair, Fast, Byzantine Fault Tolerance." Swirlds Tech Report, 2016. https://www.swirlds.com/downloads/SWIRLDS-TR-2016-01.pdf

[9] Rocket, T., Yin, M., Sekniqi, K., van Renesse, R., Sirer, E.G. "Scalable and Probabilistic Leaderless BFT Consensus through Metastability." arXiv:1906.08936, 2020. https://arxiv.org/abs/1906.08936

[10] Terry, D.B., Theimer, M.M., Petersen, K., Demers, A.J., Spreitzer, M.J., Hauser, C.H. "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System." ACM SOSP, 1995.