KONTOR OPTIMISTIC CONSENSUS

January 22, 2026

ADAM KRELLENSTEIN
adam@kontor.network

Contents

1 Introduction

Kontor Optimistic Consensus is a novel protocol for sub-block transaction confirmation in the Kontor Bitcoin metaprotocol [1] that preserves Bitcoin's position as the ultimate source of truth for transaction finality. Kontor's native currency, KOR, is staked in a virtual prediction market executed using a traditional BFT consensus algorithm to generate strong economic signals as to which transactions are *very likely* to confirm in upcoming Bitcoin blocks. This allows users to transact without waiting for Bitcoin block confirmations, with real-world latency on the order of one to two seconds.[1]

For stateful metaprotocols in general (as opposed to those that couple contract execution to the spending of particular Bitcoin transaction outputs), the ordering of transactions is semantically significant: two transactions that modify the same contract state produce different results depending on which executes first. But the order of the metaprotocol transactions within a Bitcoin block is arbitrary, as the whole block appears to the network at once.

[1]Pipelined BFT protocols such as HotStuff [2] achieve $O(n)$ message complexity per decision, allowing batch intervals of ≈1s with modest validator sets. Users wait on average half a batch interval plus one network round-trip.

Stateful metaprotocols traditionally delegate ordering to miners by executing transactions in their position within the Bitcoin block and using lexicographic ordering based on the block height and transaction index within the block.

In the Kontor system, the order of transactions within any batch expiry window is determined by a quorum of KOR stakers, who produce signed batches of transactions that assign deterministic positions to Kontor transactions before Bitcoin confirmation. When stakers sign a batch containing transaction t, they are asserting that t is valid and that no conflicting transaction will confirm on Bitcoin. If these transactions are not confirmed within the specified window, the stakers incur graduated penalties, and the transactions are simply rolled back as if a blockchain reorganization had occurred. Equivocation triggers loss of the staker's entire stake. If staker consensus fails in any way, the protocol degrades gracefully: transactions still confirm on Bitcoin and append at block-end. This strictly dominates vanilla Nakamoto pre-confirmation as an acceptance signal: for stateful metaprotocols the within-block order is arbitrary anyway, so an explicit ordering rule is required, and batches provide a deterministic ordering *before* Bitcoin confirmation. Economic finality is market-priced: each transaction specifies a minimum total bond requirement, and stakers are paid for that transaction only when the enclosing batch meets the requirement. Since Bitcoin already admits reorgs and conflict confirmations, the remaining rollback risk is not new in kind; optimistic consensus makes pre-confirmation assurance explicit, measurable, and economically accountable.

2 Model

2.1 State-Machine Replication

A blockchain operates by state-machine replication: a Byzantine fault–tolerant consensus protocol, such as Nakamoto Consensus [3], enables untrusted entities to agree on a log of events, which are executed deterministically to derive shared state. A metaprotocol extends this model with a second state machine that parses additional data from the blockchain and derives additional state.

2.2 Actors

- **Users** submit Kontor transactions. Users broadcast to both the staker set (for ordering) and to the Bitcoin network (for finality).
- **Stakers** validate transactions, produce batches, and sign orderings. Stakers bond KOR stake that is subject to slashing for protocol violations. Stakers are paid Kontor execution fees (KOR) when batched transactions finalize.
- **Recipients** query for batch inclusion, store signed batches as proofs, and accept optimistic confirmations.

2.3 Assumptions

(a) **Partial synchrony** [4]: Messages between honest stakers arrive within bound Δ after GST.
(b) **Honest supermajority:** Honest stakers hold $> \frac{2}{3}$ of total stake.
(c) **Bitcoin liveness:** Bitcoin confirms fee-paying transactions.
(d) **Bitcoin safety:** With k confirmations, the probability of blockchain reorgs is negligible.

2.4 Threat Model

We consider a Byzantine adversary $\mathcal{A}$ controlling up to 1/3 of total stake.

Adversary capabilities:

- Control Byzantine stakers (sign arbitrary messages, equivocate, or stay silent)
- Delay messages up to bound Δ after GST
- Observe all network traffic
- Coordinate across all Byzantine stakers
- Adaptively corrupt stakers up to the 1/3 bound, subject to unbonding delay

Adversary limitations:

- Cannot forge signatures (computational assumption)
- Cannot violate Bitcoin consensus
- Cannot control $\geq 1/3$ stake without detection and slashing

2.4.1 Attack Vectors

Attack	**Description**	**Defense**
Double-spend	Conflicting Bitcoin tx confirms	Kontor rollback
RBF replacement	User replaces batched tx via RBF	UTXO locking; user eats fee
Equivocation	Staker signs conflicting batches	Quorum intersection detects; slashing
Censorship	Stakers refuse to include tx	User bypasses via Bitcoin
Signature griefing	Stakers batch tx with invalid Bitcoin signature	User validates witness
Low-fee / unconfirmable batch	Stakers include txs unlikely to confirm before expiry	Fee escrow; expiry penalties; recipients treat optimistic confirmation as conditional
Low-bond batch	Quorum signs but bonds little stake	Per-tx minimum bond requirement $K_{\min}(t)$ with payment-gated rewards; stakers compete to clear higher $K_{\min}$ flow
Bond double-counting	Staker claims more bond than stake across active batches	Bond exposure invariant enforced in batch validity
Liveness	Stakers halt batch production	Graceful degradation to Bitcoin
Long-range	Old keys used after unbonding	Unbonding delay
Sybil	Many low-stake identities	Minimum stake

2.4.2 Security Goals

(a) **Safety:** No two valid batches contain conflicting transactions (under $> 2/3$ honest stake).

(b) **Liveness:** Valid transactions eventually finalize (under partial synchrony).

(c) **Accountability:** Any safety violation is attributable to $\geq 1/3$ stake with cryptographic evidence.

(d) **Incentive compatibility:** Honest behavior is a dominant strategy for rational stakers.

2.5 Notation and Parameters

The following table summarizes notation and consensus-specific parameters. For the complete protocol parameter set, see the Appendix of the Kontor Whitepaper.[1]

Symbol	Meaning	Suggested
$\mathcal{S}$	Set of stakers $\{s_1, ..., s_n\}$ (epoch-dependent)	—
σ_s	Stake of staker s	—
Σ	Total stake: $\sum_s \sigma_s$	—
$\sigma_{s,b}$	Stake bonded (risked) by staker s on batch b	—
$\Sigma_{\text{signers}}(b)$	Total stake of signers of batch b: $\sum_{s \in b.\text{signers}} \sigma_s$	—
$\Sigma_{\text{bond}}(b)$	Total bonded stake on batch b: $\sum_{s \in b.\text{signers}} \sigma_{s,b}$	—
t, t'	Transactions	—
b	Batch	—
h	Bitcoin block height	—
$f_{\text{ord}}(t)$	Ordering fee for transaction t	—
$K_{\min}(t)$	Transaction-specified minimum total bonded stake required for rewards on t	—
$\perp$	Undefined / null value	—
$:\equiv$	Definitional equality	—
$\|$	Concatenation	—
$[X]$	List of type X	—
$\{X\}$	Set of type X	—
$X \to Y$	Map from X to Y	—
n	Number of stakers	21–100
q	Quorum threshold (stake fraction)	2/3
W	Expiry window (blocks)	12
B	Target transactions per batch	~100
k	Bitcoin finality depth (confirmations)	6
Δ	Network synchrony bound	—
$\chi_{\text{consensus}}$	Fraction of emissions to consensus stakers	0.10
$\sigma_{\min}$	Minimum stake to become a staker	10M KOR
$\lambda_{\text{bond,min}}$	Minimum bond as fraction of stake when signing a batch	0.001
λ_{equiv}	Equivocation penalty (fraction of stake)	1.00
λ_{invalid}	Invalid batch penalty (fraction of stake)	0.10
$\lambda_{\text{conflict}}$	Conflict penalty (fraction of stake)	0.10
σ_{expiry}	Expiry base penalty per transaction	10K KOR
λ_{cap}	Max bonded stake lost per batch	0.01
β_{slash}	Slash burn rate (rest to reporter)	0.50
β_{fee}	Service fee burn rate (ordering and bundling)	0.50

2.6 Protocol Objects

2.6.1 Primitive Types

We define the following primitive types:

Type	Representation	Description
TxId	`bytes32`	Transaction identifier (hash of signed transaction)
StakerId	`bytes32`	Staker identifier (public key hash)
Signature	`bytes64`	ECDSA or Schnorr signature
BlockHash	`bytes32`	Bitcoin block hash
BlockHeight	$\mathbb{N}$	Bitcoin block height
BatchId	$\mathbb{N}$	Monotonically increasing batch identifier
Epoch	$\mathbb{N}$	Staker set epoch number
Index	$\mathbb{N}$	Position within batch
Stake	$\mathbb{N}$	Stake amount (in base units)

2.6.2 Transactions

A Kontor transaction wraps a Bitcoin transaction with application-specific payload:

$$\text{Transaction} := \begin{cases} \text{id} & : \text{TxId} \\ \text{inputs} & : \{\text{UTXO}\} \\ \text{outputs} & : [\text{Output}] \\ \text{witness} & : \text{Bytes} \\ \text{payload} & : \text{KontorPayload} \end{cases} \tag{1}$$

where $\{X\}$ denotes a set of X and $[X]$ denotes a list of X.

Conflict Relation: Two transactions *conflict* if they spend at least one common input:

$$\text{Conflict}(t_1, t_2) :\equiv t_1.\text{inputs} \cap t_2.\text{inputs} \neq \emptyset \tag{2}$$

This captures Bitcoin's double-spend semantics: conflicting transactions cannot both confirm on Bitcoin.

2.6.3 Batches

A batch is an ordered collection of transactions with quorum attestation:

$$\text{Batch} := \begin{cases} \text{batch\_id} & : \text{BatchId} \\ \text{epoch} & : \text{Epoch} \\ \text{chain\_tip} & : \text{BlockHash} \\ \text{expiry} & : \text{BlockHeight} \\ \text{txs} & : [\text{Transaction}] \\ \text{signers} & : [\text{StakerId}] \\ \text{bonds} & : [\text{Stake}] \\ \text{signatures} & : [\text{Signature}] \end{cases} \tag{3}$$

The `bonds` list is aligned with `signers`: `b.bonds[i]` is the amount risked by `b.signers[i]` on batch b. Formally, for staker $s = b.\text{signers}[i]$:

$$\sigma_{s,b} :\equiv b.\text{bonds}[i] \tag{4}$$

Bond collateralization: Bonds are stake-at-risk declarations. The protocol enforces that the total stake-at-risk implied by a staker's bonds on unresolved batches is fully collateralized by their total stake.

$$\text{BondExposure}(s, \text{state}) :\equiv \sum_{b \in \text{ ActiveBatches(state)}: s \in b.\text{signers}} \sigma_{s,b} \tag{5}$$

$$\text{ActiveBatches(state)} := \{b \in \text{state.batches} : \exists t \in b.\text{txs} : \neg \text{ IsResolved}(t, \text{state})\} \tag{6}$$

Bond exposure invariant: A batch is valid only if each signer remains fully collateralized after adding this batch's bond:

$$\forall s \in b.\text{signers} : \text{BondExposure}(s, \text{state}) + \sigma_{s,b} \leq \sigma_s \tag{7}$$

This prevents over-promising: a staker cannot advertise more maximum per-batch stake-at-risk across unresolved batches than they actually have.

Batch sequence and dependencies: Batches are decisions of a BFT state-machine replication protocol and therefore form a single ordered log, indexed by `batch_id`. Batches may be produced faster than Bitcoin finalizes earlier batches; as a result, it is normal for nodes to observe batches $b, b+1, b+2, ...$ while the Bitcoin outcomes for earlier positions are still unresolved. A batch can be structurally valid and have a valid quorum signature regardless of whether earlier batches ultimately expire or are rolled back; however, transaction **finalization** is conditional on the resolution of all prior positions (see Execution Blocking). Nodes may still execute batched transactions optimistically before Bitcoin confirmation. In particular, if a transaction at some earlier position is rolled back due to a Bitcoin-confirmed conflict, all subsequent positions—including those in later batches—are rolled back as well (cascading rollback).

The `chain_tip` field commits the batch to a specific Bitcoin state. Stakers sign only if all transaction inputs are unspent as of that block. This prevents using a stale batch to double-spend after Bitcoin confirmation: once a conflicting transaction confirms on Bitcoin, the chain tip advances and the batch is invalidated.

Invariants:

- $|b.\text{signers}| = |b.\text{bonds}| = |b.\text{signatures}|$ — each signer provides one bond + one signature
- $\forall i \neq j : \neg \text{ Conflict}(b.\text{txs}[i], b.\text{txs}[j])$ — no internal conflicts
- b.txs defines a total order on included transactions
- $\forall t \in b.\text{txs} : \text{InputsUnspent}(t, b.\text{chain\_tip})$ — all inputs unspent at chain tip

where $\text{InputsUnspent}(t, h) :\equiv \forall \text{ input} \in t.\text{inputs} : \neg\exists \text{tx}' \in \text{TxsConfirmedBy}(h) : \text{input} \in \text{tx}'.\text{inputs}$

and $\text{TxsConfirmedBy}(h) := \{t : \text{state.bitcoin\_confirmed}[t.\text{id}] \leq \text{HeightOf}(h)\}$ is the set of transactions confirmed at or before block h.

Position: A transaction's position is its unique location in the ordering:

$$\text{Position} := (\text{BatchId}, \text{Index}) \tag{8}$$

$$\text{Position}(t, b) := (b.\text{batch\_id}, \text{IndexOf}(t, b.\text{txs})) \tag{9}$$

Positions are totally ordered lexicographically: $(b_1, i_1) < (b_2, i_2) \Leftrightarrow b_1 < b_2 \vee (b_1 = b_2 \wedge i_1 < i_2)$.

2.6.4 Staker Set

The staker set defines the active validators for an epoch:

$$\text{StakerSet} := \begin{cases} \text{epoch} & : \text{Epoch} \\ \text{stakers} & : \text{StakerId} \to \text{Stake} \end{cases} \tag{10}$$

Total Stake:

$$\text{TotalStake}(S) \coloneqq \sum_{s \in \text{ dom}(S.\text{stakers})} S.\text{stakers}[s] \tag{11}$$

Quorum Stake: The minimum stake required for a valid quorum:

$$\text{QuorumStake}(S) \coloneqq \left\lceil \frac{2}{3} \times \text{TotalStake}(S) \right\rceil \tag{12}$$

Valid Quorum: A set of signers forms a valid quorum if their combined stake meets the threshold:

$$\text{ValidQuorum}(\text{signers}, S) :\equiv \sum_{s \in \text{ signers}} S.\text{stakers}[s] \geq \text{QuorumStake}(S) \tag{13}$$

Quorum Intersection Property: Any two valid quorums overlap in at least $\frac{1}{3}$ of total stake:

$$\forall Q_1, Q_2 \text{ valid quorums} : \sum_{s \in Q_1 \cap Q_2} S.\text{stakers}[s] \geq \frac{1}{3} \cdot \text{TotalStake}(S) \tag{14}$$

This property is fundamental to safety: if honest stakers hold $> \frac{2}{3}$ stake, every two quorums share at least one honest staker.

2.6.5 Signature Verification

Batch Digest: The canonical message to sign for a batch:

$$\begin{aligned} \text{BatchDigest}(b) \coloneqq \text{Hash}(&b.\text{batch\_id} \parallel b.\text{epoch} \parallel b.\text{chain\_tip} \\ &\parallel b.\text{expiry} \parallel \text{MerkleRoot}(b.\text{txs})) \end{aligned} \tag{15}$$

where $\parallel$ denotes concatenation and MerkleRoot computes the Merkle root of transaction IDs.

Bonded Digest: Each signer commits to their chosen bond amount in the signed message:

$$\text{BondedDigest}(b, \beta) \coloneqq \text{Hash}(\text{BatchDigest}(b) \parallel \beta) \tag{16}$$

where β is the bond value.

Signature Validity:

$$\text{ValidSignature}(\text{sig}, \text{signer}, \beta, b) :\equiv \text{Verify}(\text{sig}, \text{BondedDigest}(b, \beta), \text{PublicKey}(\text{signer})) \tag{17}$$

Witness Validity: A transaction has a valid witness if its signature data correctly authorizes spending of all inputs:

$$\text{ValidWitness}(t) :\equiv \forall \text{ input} \in t.\text{inputs} : \text{VerifyScript}(\text{input.scriptPubKey}, t, t.\text{witness}) \tag{18}$$

Batch Signatures: A batch has valid signatures if:

$$\begin{aligned} \text{ValidBatchSignatures}(b, S) :\equiv\ & |b.\text{signers}| = |b.\text{bonds}| = |b.\text{signatures}| \\ & \wedge \forall i : \text{ValidSignature}(b.\text{signatures}[i], b.\text{signers}[i], b.\text{bonds}[i], b) \\ & \wedge \forall i : \\ & \quad \wedge b.\text{bonds}[i] \leq S.\text{stakers}[b.\text{signers}[i]] \\ & \wedge \forall s \in b.\text{signers} : s \in \text{dom}(S.\text{stakers}) \\ & \wedge \text{ValidQuorum}(b.\text{signers}, S) \end{aligned} \qquad b.\text{bonds}[i] \geq \lambda_{\text{bond,min}} \cdot S.\text{stakers}[b.\text{signers}[i]] \tag{19}$$

2.6.6 Batch Validity

A batch is valid if it satisfies structural constraints, has valid quorum signatures, and its chain tip is on the canonical chain:

$$
\begin{aligned}
\text{ValidBatch}(b, S, \text{state}) :\equiv\ & b.\text{epoch} = S.\text{epoch} \\
& \wedge\ b.\text{chain\_tip} \in \text{state.canonical\_history} \\
& \wedge\ \text{ValidBatchSignatures}(b, S) \\
& \wedge\ \forall s \in b.\text{signers} : \text{BondExposure}(s, \text{state}) + \sigma_{s,b} \leq \sigma_s \\
& \wedge\ \forall t \in b.\text{txs} : \text{ValidTransaction}(t, \text{state}) \\
& \wedge\ \forall i \neq j : \neg\ \text{Conflict}(b.\text{txs}[i], b.\text{txs}[j])
\end{aligned}
\tag{20}
$$

Transaction Validity: A transaction is valid if it passes all validation checks:

$$
\begin{aligned}
\text{ValidTransaction}(t, \text{state}) :\equiv\ & \text{InputsUnspent}(t, \text{state}) \\
& \wedge\ \text{ValidPayload}(t.\text{payload}) \\
& \wedge\ \text{ValidWitness}(t)
\end{aligned}
\tag{21}
$$

Fee policy: Whether a transaction is **worth batching** depends on its Bitcoin fee rate relative to current network conditions. This is not a consensus-validity rule: it is a local policy decision by stakers/recipients.

Minimum bond requirement: Each transaction specifies a minimum total bond requirement $K_{\min}(t)$. This is not a batch validity rule; it is a payment gating rule for ordering rewards (see Staker Rewards).

2.6.7 Global State

The global protocol state separates Bitcoin state from ordering state:

$$
\text{GlobalState} := \begin{cases}
\text{block\_height} & : \text{BlockHeight} \\
\text{bitcoin\_confirmed} & : \text{TxId} \to \text{BlockHeight} \\
\text{canonical\_history} & : \{\text{BlockHash}\} \\
\text{batches} & : [\text{Batch}] \\
\text{staker\_sets} & : \text{Epoch} \to \text{StakerSet} \\
\text{kontor\_state} & : \text{KontorState}
\end{cases}
\tag{22}
$$

The `kontor_state` field contains application state (account balances, contract storage, etc.) computed deterministically from executed transactions. The `canonical_history` field tracks all block hashes that were part of the canonical Bitcoin chain (for reorg detection). Transaction validation checks UTXOs against this state.

Block Height Lookup: We define $\text{HeightOf} : \text{BlockHash} \to \text{BlockHeight}$ as the lookup function that returns the height of a block given its hash. This is well-defined for any block in `canonical_history`.

3 Protocol

3.1 Operations

3.1.1 Transaction Submission

A user submits a transaction by broadcasting to all stakers:

Algorithm 1: Transaction Submission

```
1: function SUBMIT(tx)
2:     for s ∈ 𝒮 do
3:         result ← Send(tx, s)
4:     end
5: end
```

3.1.2 Transaction Validation

Stakers validate transactions before adding them to the pending pool:

Algorithm 2: Transaction Validation

```
function VALIDATE(tx, state)
    ▷ Check UTXO validity
    if ∃ input ∈ tx.inputs : ¬ Unspent(input, state) then
        return false
    end

    ▷ Check payload validity
    if ¬ ValidPayload(tx.payload) then
        return false
    end

    ▷ Check Bitcoin witness validity
    if ¬ ValidWitness(tx) then
        return false
    end

    ▷ Check not already batched
    if ∃b ∈ state.batches : tx ∈ b.txs then
        return false
    end

    ▷ Check no conflicts with batched transactions
    if ∃b ∈ state.batches : ∃t′ ∈ b.txs : Conflict(tx, t′) then
        return false
    end
    return true
end
```

The validation rules ensure:

(a) **UTXO validity:** All inputs are unspent in current Kontor state
(b) **Payload validity:** The Kontor payload parses and is well-formed
(c) **Witness validity:** Bitcoin signatures and scripts are valid (so batched transactions can actually confirm on Bitcoin)
(d) **Fee policy:** Stakers and recipients apply local policy based on Bitcoin miner fee rate (e.g., rejecting transactions unlikely to confirm before expiry)
(e) **Uniqueness:** Transaction not already batched
(f) **No conflicts:** No existing batched transaction conflicts

3.1.3 Batch Formation

Stakers periodically form batches via BFT consensus. A batch is an ordered list of non-conflicting transactions with a committed chain tip and expiry block. The leader selects transactions from the pending pool (prioritized by fee, filtered for conflicts), constructs a proposal, and initiates BFT voting. Upon quorum ($> \frac{2}{3}$ stake), the batch is signed and published. Each batch assigns deterministic positions to transactions before Bitcoin confirmation, enabling sub-block finality.

Algorithm 3: Batch Formation

```
function FORMBATCH(pending, state)
    ▷ Select non-conflicting transactions from pending pool
    txs ← Select(pending, MaxBatchSize)

    ▷ Construct batch proposal
```

```
    batch ← (batch_id : NextBatchId(state), epoch : CurrentEpoch(state), chain_tip :
    LatestBlockHash(state), expiry : state.block_height + W, txs : txs, signers :
    [], bonds : [], signatures : [])

    ▷ Run BFT consensus to get quorum signatures
    return BFTConsensus(batch)
end
```

The Select function chooses transactions from the pending pool, respecting the constraint that no two selected transactions conflict.

3.1.4 Batch Signing

Each staker in the quorum signs valid batch proposals:

Algorithm 4: Batch Signing

```
function SignBatch(batch, staker)
    if ValidBatchProposal(batch) then
        bond ← ChooseBond(batch, staker)
        sig ← Sign(BondedDigest(batch, bond), PrivateKey(staker))
        return (staker.id, bond, sig)
    end
    return ⊥
end
```

Honest Staker Rule: An honest staker signs a batch proposal only if:

(a) The batch is well-formed

(b) The `chain_tip` is on the canonical chain and all transaction inputs are unspent as of that block

(c) No transaction in the batch conflicts with any previously-signed batch

(d) The batch epoch matches the staker's current epoch

3.1.5 Batch Publication

Signed batches are broadcast to all nodes:

Algorithm 5: Batch Publication

```
function PublishBatch(batch)
    for n ∈ Nodes do
        result ← Send(batch, n)
    end
end
```

3.1.6 Recipient Verification

Recipients verify batches before accepting optimistic confirmations:

Algorithm 6: Optimistic Confirmation Verification

```
function VerifyOptimisticConfirmation(tx, batch, state)
    ▷ Verify transaction is in batch
    if tx ∉ batch.txs then
        return false
    end

    ▷ Verify batch has valid quorum signatures
    if ¬ ValidBatch(batch, state.staker_sets[batch.epoch], state) then
        return false
    end
```

```
11:
12:   ▷ Verify batch has not expired
13:   if batch.expiry ≤ state.block_height then
14:     return false
15:   end
16:
17:   ▷ Verify chain_tip is on canonical chain and recent
18:   if batch.chain_tip ∉ state.canonical_history ∨ HeightOf(batch.chain_tip) <
      state.block_height − W then
19:     return false
20:   end
21:
22:   ▷ Verify Bitcoin transaction signatures are valid
23:   for t ∈ batch.txs do
24:     if ¬ ValidWitness(t) then
25:       return false
26:     end
27:   end
28:   return true
29: end
```

Recipients should store the signed batch locally as evidence of the ordering.

3.1.7 Bitcoin Broadcast

Users broadcast transactions to Bitcoin independently:

Algorithm 7: Bitcoin Broadcast

```
1: function BroadcastBitcoin(tx)
2:   result ← BitcoinBroadcast(tx)
3: end
```

The user controls when and whether to broadcast. This preserves UTXO sovereignty.

3.2 Conflict Resolution

Bitcoin is the ultimate arbiter of transaction validity. When a Bitcoin-confirmed transaction conflicts with a batched transaction (e.g., a double-spend or RBF replacement), the batched transaction is rolled back and stakers are slashed. This algorithm handles three cases: (1) the batched transaction itself confirms—finalize it at its assigned position, (2) a conflicting transaction confirms—trigger cascading rollback of the batched transaction and all subsequent positions, or (3) an unbatched transaction confirms—append it at block-end.

When a transaction t_B confirms on Bitcoin at height h:

Algorithm 8: Conflict Resolution

```
1: function Resolve(t_B, h, state)
2:   ▷ Check if this is the batched transaction confirming
3:   if WasBatched(t_B, state) then
4:     ▷ Batched transaction confirmed; finalize at its batch position
5:     result ← Finalize(t_B, h)
6:     return
7:   end
8:
9:   ▷ Find any conflicting batched transaction
10:  t_A ← FindConflictingBatchedTx(t_B, state)
11:
12:  if t_A ≠ ⊥ then
13:    ▷ Conflicting tx confirmed; rollback t_A and all subsequent txs
```

```
    result ← CascadingRollback(t_A, state)
  end

  ▷ Append Bitcoin-Confirmed tx at block-end
  result ← Append(t_B, h)
end
```

Cascading rollback: When transaction t_A at position (b, i) is rolled back, all transactions at positions $\geq (b, i)$ must also be rolled back. State changes cannot be selectively undone—execution is sequential, and later transactions may depend on earlier ones.

3.2.1 Block-End Ordering

Unbatched transactions are appended at "block-end" with a deterministic tie-breaker.

Batched transactions have positions of the form $(\text{BatchId}, \text{Index})$. Unbatched transactions use a separate ordering domain: for an unbatched transaction confirmed at height h with index i within that block, its position is $(\text{BlockEnd}, h, i)$.

Global ordering: Execution proceeds in two phases: (1) all batched transactions execute in batch order (by `batch_id`, then index within batch), then (2) unbatched transactions execute ordered by (h, i) where h is the Bitcoin confirmation height and i is the transaction's index within block h. This ensures all indexers derive identical ordering from Bitcoin data alone.

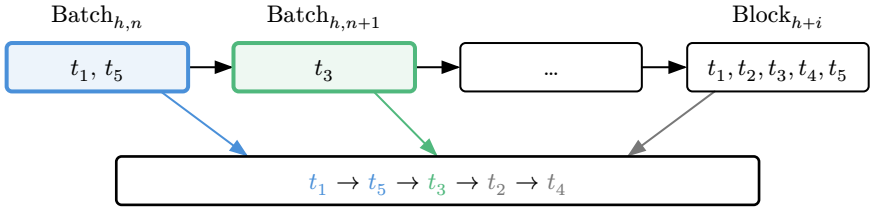

FIGURE 1. **Execution Order**: Batched transactions (Batch$_{h,n}$, Batch$_{h,n+1}$) execute first in batch order, followed by unbatched transactions (gray) at block-end.

3.2.2 Conflict Search

Algorithm 9: Conflict Search

```
function FindConflictingBatchedTx(t_B, state)
  for b ∈ state.batches do
    for t_A ∈ b.txs do
      if Conflict(t_A, t_B) then
        return t_A
      end
    end
  end
  return ⊥
end
```

3.2.3 RBF Handling

Bitcoin's Replace-By-Fee (RBF) mechanism allows users to replace unconfirmed transactions with higher-fee alternatives. This creates an attack vector: a user could submit a transaction, receive optimistic confirmation via batching, then broadcast an RBF replacement before Bitcoin confirmation.

UTXO Locking: The protocol prevents this by treating batched UTXOs as spent for validation purposes. The "No conflicts" validation rule (Section 3.1.2) rejects any Kontor transaction that spends inputs already claimed by a pending batched transaction. Indexers

must process batches (which propagate via the staker network) in addition to Bitcoin blocks to maintain consistent state; batch signatures make them self-authenticating.

If a user broadcasts an RBF replacement on Bitcoin after their transaction has been batched:

- A conflicting transaction can confirm on Bitcoin instead of the batched transaction
- The batched transaction is **Rolled back** (and any optimistic execution is reverted via cascading rollback)
- The user forfeits their Kontor execution fee (KOR) if stakers submit a griefing proof; otherwise signers are slashed under the conflict rule

This places the protocol-level penalty for RBF replacement on the user who initiated it, rather than on honest stakers.

Signature Griefing Defense: UTXO locking would be unsafe if batches could include transactions with invalid Bitcoin signatures: such transactions would never confirm, but could lock UTXOs until expiry. Kontor prevents this by making $\text{ValidWitness}(t)$ part of transaction and batch validity. Invalid-witness transactions are rejected and do not trigger UTXO locking. Recipients may still validate witnesses independently before accepting an optimistic confirmation.

3.2.4 Cross-Batch Conflict Tiebreaker

Under honest majority, safety prevents conflicting transactions in different batches. This tiebreaker handles Byzantine scenarios where safety is violated:

Algorithm 10: Cross-Batch Tiebreaker

```
function TIEBREAKER(t_A, t_B)
    (b_A, i_A) ← Position(t_A)
    (b_B, i_B) ← Position(t_B)

    if b_A < b_B then
        return t_A
    end
    if b_B < b_A then
        return t_B
    end

    ▷ Same batch: use index
    if i_A < i_B then
        return t_A
    end
    return t_B
end
```

This deterministic tiebreaker ensures all nodes resolve conflicts identically, even in the presence of Byzantine behavior.

3.3 State Execution

A transaction progresses through distinct states before it becomes **Batch-Confirmed**. Understanding these states is critical to the protocol's semantics.

3.3.1 Transaction States

State	Definition	Guarantee
Batched	Included in a signed batch with a position	Economic finality: $\geq 2/3$ stake attests to ordering
Bitcoin-Confirmed	Confirmed on Bitcoin	Bitcoin finality (probabilistic)

State	Definition	Guarantee
Batch-Confirmed	Batched AND Bitcoin-Confirmed	Full finality: ordering + Bitcoin
Expired	Batch expiry passed without confirmation	Stakers penalized; may re-batch
Rolled back	Conflicting tx confirmed on Bitcoin	Stakers slashed (unless griefing proof)

3.3.2 Execution Order

Kontor state executes strictly by position:

Algorithm 11: State Execution

```
1:  function ExecuteState(state)
2:      ▷ Collect all executable transactions
3:      pending ← CollectExecutableTxs(state)
4:
5:      ▷ Sort by position for deterministic ordering
6:      ordered ← SortByPosition(pending)
7:
8:      ▷ Apply transactions in order
9:      for tx ∈ ordered do
10:         result ← ApplyToKontorState(tx, state.kontor_state)
11:     end
12: end
```

3.3.3 Position Ordering

Algorithm 12: Position Ordering

```
1:  function SortByPosition(txs)
2:      ▷ Sort transactions lexicographically by (batch_id, index)
3:      return Sort(txs, (t_A, t_B) ⇒ Position(t_A) < Position(t_B))
4:  end
```

3.3.4 Finalization Criteria

A batched transaction is **Batch-Confirmed** when it is both batched and the same transaction confirmed on Bitcoin:

$$\begin{aligned}\text{IsBatchConfirmed}(t, \text{state}) :\equiv\ & \text{WasBatched}(t, \text{state}) \\ & \wedge\ \text{IsBitcoinConfirmed}(t, \text{state}) \\ & \wedge\ \neg\ \text{IsRolledBack}(t, \text{state})\end{aligned} \tag{23}$$

$$\text{IsBitcoinConfirmed}(t, \text{state}) :\equiv t.\text{id} \in \text{state.bitcoin\_confirmed} \tag{24}$$

$$\begin{aligned}\text{IsRolledBack}(t, \text{state}) :\equiv\ & \text{WasBatched}(t, \text{state}) \\ & \wedge\ \exists t' \in \text{state.bitcoin\_confirmed} : t' \neq t \wedge \text{Conflict}(t, t')\end{aligned} \tag{25}$$

3.3.5 Execution Blocking

Nodes execute batched transactions optimistically (before Bitcoin confirmation). Execution is blocked only when the ordering prefix is unavailable (e.g., missing batch data).

$$\begin{aligned}\text{CanExecute}(t, \text{state}) :\equiv\ & \text{let } (b, i) = \text{Position}(t) \\ & \wedge\ \text{WasBatched}(t, \text{state}) \\ & \wedge\ \forall p \in \text{PriorPositions}(b, i) : \text{IsKnown}(p, \text{state})\end{aligned} \tag{26}$$

Finalization is sequential: a position cannot become fully final until all prior positions have resolved on Bitcoin.

$$\begin{aligned}\text{CanFinalize}(t, \text{state}) :\equiv \text{let } (b, i) = \text{Position}(t) \\ \wedge\ \text{IsBatchConfirmed}(t, \text{state}) \\ \wedge\ \forall p \in \text{PriorPositions}(b, i) : \text{IsResolved}(\text{TxAt}(p, \text{state}), \text{state})\end{aligned} \tag{27}$$

$$\begin{aligned}\text{IsKnown}(p, \text{state}) :\equiv \text{let } (b, i) = p \\ \wedge\ b < |\text{state.batches}| \\ \wedge\ i < |\text{state.batches}[b].\text{txs}|\end{aligned} \tag{28}$$

$$\begin{aligned}\text{IsResolved}(t, \text{state}) :\equiv (\text{IsBatchConfirmed}(t, \text{state}) \\ \vee\ \text{IsExpired}(t, \text{state}) \\ \vee\ \text{IsRolledBack}(t, \text{state}))\end{aligned} \tag{29}$$

This ensures deterministic, position-ordered execution while allowing low-latency optimistic operation; Bitcoin confirmation only gates finality.

3.3.6 Auxiliary State Functions

The following functions complete the state execution semantics:

Prior Positions: All positions that must resolve before (b, i):

$$\text{PriorPositions}(b, i) \coloneqq \{(b', i') : b' < b\} \cup \{(b, i') : i' < i\} \tag{30}$$

Transaction Lookup: Retrieve transaction at a given position:

$$\text{TxAt}((b, i), \text{state}) \coloneqq \text{state.batches}[b].\text{txs}[i] \tag{31}$$

Batch Lookup: Find the batch containing a transaction:

$$\text{BatchOf}(t, \text{state}) \coloneqq b \text{ where } t \in b.\text{txs for some } b \in \text{state.batches} \tag{32}$$

Expiration: A batched transaction expires if its batch expiry passes without Bitcoin confirmation:

$$\begin{aligned}\text{IsExpired}(t, \text{state}) :\equiv \text{let } b = \text{BatchOf}(t, \text{state}) \\ \wedge\ \neg\ \text{IsBitcoinConfirmed}(t, \text{state}) \\ \wedge\ \neg\ \text{IsRolledBack}(t, \text{state}) \\ \wedge\ \text{state.block\_height} \geq b.\text{expiry}\end{aligned} \tag{33}$$

Batched Check: Whether a transaction was included in any batch:

$$\text{WasBatched}(t, \text{state}) :\equiv \exists b \in \text{state.batches} : t \in b.\text{txs} \tag{34}$$

3.4 Implementation Requirements

3.4.1 Epoch Transitions

Epochs partition time into periods with fixed staker sets. Any epoch mechanism must satisfy:

Duration: Epochs have bounded duration $E_{\min} \leq |e| \leq E_{\max}$ in Bitcoin blocks.

Determinism: Epoch boundaries are deterministically computable from Bitcoin state.

Overlap: For safety across transitions, batches signed in epoch e remain valid for W blocks into epoch $e + 1$, ensuring pending batches can finalize.

Unbonding delay: Stakers must wait ≥ 1 epoch after requesting unbonding before withdrawal. This prevents long-range attacks where an attacker unbonds, then uses old keys to sign conflicting batches.

Quorum continuity: Honest stake $> \frac{2}{3}$ must hold in every epoch. Staker set changes must preserve this invariant.

3.4.2 Timing Parameters

Timing parameters must satisfy the following relationships:

$$\text{BatchInterval} < \frac{W}{2} \tag{35}$$

where W is the expiry window. This ensures transactions have time to confirm after batching.

$$\text{UnbondingDelay} > W \tag{36}$$

Stakers remain slashable for the full expiry window of any batch they signed.

Confirmation timing:

$$\text{OptimisticConfirmationTime} \approx \text{BatchInterval} + \Delta \tag{37}$$

$$\text{FinalConfirmationTime} \approx \text{OptimisticConfirmationTime} + k \times \beta \tag{38}$$

where $\beta \approx 600$s is Bitcoin block time and k is finality depth.

3.4.3 BFT Consensus

The protocol requires any BFT consensus satisfying:

Safety: No two conflicting batches receive quorum signatures (under $> \frac{2}{3}$ honest stake).

Liveness: Valid transactions are eventually included in batches (under partial synchrony).

Accountability: Safety violations produce cryptographic evidence attributable to $\geq \frac{1}{3}$ stake.

Stake-weighted voting: Votes and quorums are weighted by each staker's total stake σ_s in the epoch staker set. The per-batch bond $\sigma_{s,b}$ does not affect BFT voting power; it only determines (i) how consensus emissions and batch fees are paid, and (ii) how much stake is at risk for non-fatal batch failures. This separation preserves standard BFT safety assumptions while letting the market price economic finality.

Minimum stake: The minimum stake $\sigma_{\min}$ bounds the number of consensus participants and mitigates Sybil attacks on the network layer. It is orthogonal to bond sizing: even a high-stake staker can choose to bond less on a particular batch. Transactions express desired economic finality by setting $K_{\min}(t)$, and stakers must clear those thresholds to earn rewards.

Standard protocols (PBFT [5], HotStuff [2], Tendermint [6]) satisfy these properties. The choice is an implementation detail.

3.4.4 Network Layer

Batch propagation: Signed batches reach all honest nodes within bounded time Δ.

Batch retrieval: Any staker can serve batch data; quorum signatures make batches self-authenticating.

Fraud proofs: If stakers withhold a signed batch, the batch in the recipient's possession is the fraud proof.

3.5 Protocol Flows

The normal protocol flow:

(a) User submits transaction t to stakers
(b) Stakers validate t and include it in the next batch via BFT consensus
(c) Batch b is signed by a quorum ($\geq \frac{2}{3}$ stake) and published
(d) Transaction t receives position $(b.\text{batch\_id}, i)$ in the global ordering
(e) User broadcasts t to Bitcoin
(f) When t confirms on Bitcoin, t becomes Batch-Confirmed; if a conflict confirms, t is rolled back

If the staker set fails, users can still transact via Bitcoin directly.

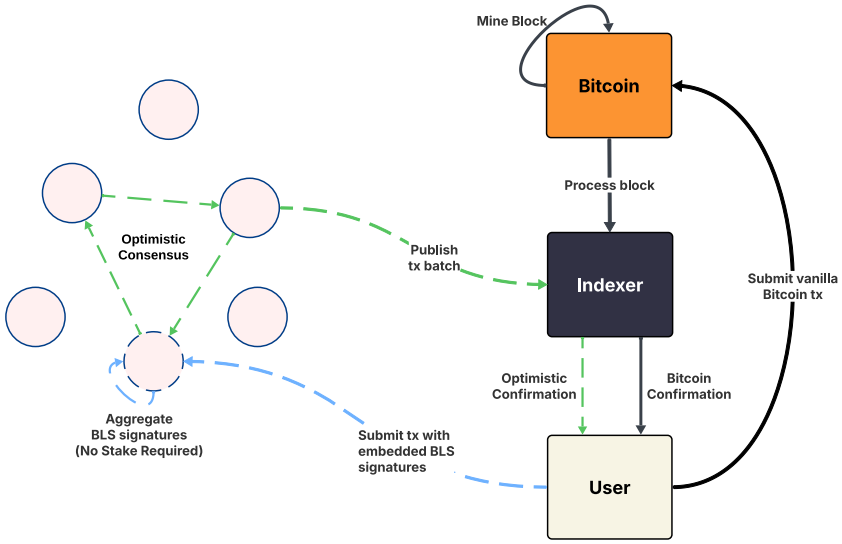

FIGURE 2. Normal Protocol Flow

3.5.1 Outcome Matrix

Scenario	Kontor Payload	Stakers	User
Batched, confirms	Batch-Confirmed at position (b, i)	Paid	Success
Not batched, confirms	Appended at block-end	N/A	Success (no ordering)
Batched t_A, conflicting t_B confirms	t_A + subsequent txs rolled back	Slashed	Double-spend; rollback
Batched, expires	Expired	Not paid	May re-batch

4 Incentives & Economics

4.1 Stake Requirements

Consensus stakers must lock KOR to participate. This stake:

- Provides the capital backing for ordering guarantees
- Creates skin-in-the-game for honest behavior
- Enables slashing for protocol violations

Consensus staking is part of Kontor's unified staking model: the same KOR can simultaneously back storage commitments, consensus participation, and bridge operations. An operator's total stake must exceed the sum of their commitments across all roles; slashing in one role affects capacity in all others. (See the Kontor Whitepaper. [1])

4.1.1 Minimum Stake

$$\sigma_s \geq \sigma_{\min} \tag{39}$$

The minimum stake $\sigma_{\min}$ bounds the maximum number of stakers. With 1B total KOR supply and the suggested $\sigma_{\min}$, each staker would lock at least 1% of supply, implying a theoretical maximum of 100 stakers.

4.1.2 Staker Set

The set of consensus stakers at epoch e is denoted $\mathcal{S}$. Each staker $s \in \mathcal{S}$ has stake $\sigma_s \geq \sigma_{\min}$.

Total consensus stake: $\Sigma \stackrel{\text{def}}{=} \sum_{s \in \mathcal{S}} \sigma_s$

4.2 Staker Rewards

Stakers earn revenue from two sources: protocol emissions and ordering fees. For a unified view of staking yields across all protocol services, see the Kontor Staking Yield Analysis.[7]

4.2.1 Emission Share

A fraction $\chi_{\text{consensus}}$ of total KOR emissions is allocated to consensus stakers. Per block h:

$$\varepsilon_{\text{consensus}}(h) \stackrel{\text{def}}{=} \varepsilon(h) \cdot \chi_{\text{consensus}} \tag{40}$$

where $\varepsilon(h)$ is total network emissions per block (see Kontor Whitepaper [1] for the emission schedule).

Emissions are distributed to stakers who successfully provide optimistic-consensus service. Let $\text{EligibleConfirmedTxs}(b, h)$ be the subset of transactions in batch b that become Batch-Confirmed at Bitcoin height h and whose minimum bond requirement is met:

$$\text{EligibleConfirmedTxs}(b, h) \coloneqq \{t \in \text{ConfirmedTxs}(b, h) : \Sigma_{\text{bond}}(b) \geq K_{\min}(t)\} \tag{41}$$

Let $\mathcal{B}_{\text{succ}}(h)$ be the set of batches for which at least one eligible transaction becomes Batch-Confirmed at Bitcoin height h:

$$\mathcal{B}_{\text{succ}}(h) \coloneqq \{b : |\text{EligibleConfirmedTxs}(b, h)| > 0\} \tag{42}$$

Define the bonded stake weight for height h:

$$w_{s(h)} :\equiv \sum_{b \in \mathcal{B}_{\text{succ}}(h) : s \in b.\text{signers}} |\text{EligibleConfirmedTxs}(b, h)| \cdot \sigma_{s,b} \tag{43}$$

$$W(h) :\equiv \sum_{s \in \mathcal{S}} w_{s(h)} \tag{44}$$

Then emissions are distributed proportionally to bonded stake weight (zero if $W(h) = 0$):

$$r_{\text{emission}}(s, h) \stackrel{\text{def}}{=} \begin{cases} \varepsilon_{\text{consensus}}(h) \cdot \left(\frac{w_{s(h)}}{W(h)}\right) & \text{if } W(h) > 0 \\ 0 & \text{otherwise} \end{cases} \tag{45}$$

This makes emissions a payment for successful optimistic confirmations, avoids paying for unresolved batches, and preserves the interpretation of $\sigma_{s,b}$ as stake-at-risk per batch.

4.2.2 Ordering Fees

Ordering stakers are funded primarily by emissions, which cover the base cost of providing optimistic consensus. The ordering fee $f_{\text{ord}}(t)$ is an optional payment that users can include to increase priority or express willingness to pay for service.

Ordering fees are escrowed when a transaction is included in a signed batch. On Batch-Confirmed, fees are paid to signers; on expiry or rollback, fees are burned.

Let $\text{ConfirmedTxs}(b, h)$ be the subset of $b.\text{txs}$ that become Batch-Confirmed at Bitcoin height h. Ordering fee payouts are payment-gated per transaction: if $\Sigma_{\text{bond}}(b) < K_{\min}(t)$, then t's non-burned fee portion is not paid to signers (it is burned). A fraction β_{fee} of fees is burned even on success; the remainder of eligible fees is distributed to signers proportionally to bonded stake:

$$r_{\text{fee}}(s, b, h) \stackrel{\text{def}}{=} (1 - \beta_{\text{fee}}) \cdot \left(\sum_{t \in \text{ EligibleConfirmedTxs}(b,h)} f_{\text{ord}}(t) \right) \cdot \left(\frac{\sigma_{s,b}}{\Sigma_{\text{bond}}}(b) \right) \tag{46}$$

where $\Sigma_{\text{bond}}(b) = \sum_{s \in b.\text{signers}} \sigma_{s,b}$.

4.2.3 Total Staker Revenue

$$r_{\text{staker}}(s, h) \stackrel{\text{def}}{=} r_{\text{emission}}(s, h) + \sum_{b \in \mathcal{B}_{\text{succ}}(h)} r_{\text{fee}}(s, b, h) \tag{47}$$

4.2.4 Dynamic Bond Sizing (Prediction-Market Designs)

Kontor treats optimistic consensus as a virtual prediction market on the event this batch confirms on Bitcoin before expiry without conflict. Each signer chooses how much stake to bond to the batch, $\sigma_{s,b}$, trading off higher expected reward against higher loss if the batch expires or is rolled back.

Design A: Free-market bonding (yield clears).

- The protocol fixes the consensus emission stream $\varepsilon_{\text{consensus}}(h)$.
- Stakers allocate $\sigma_{s,b}$ batch-by-batch. Users express their desired economic finality by setting $K_{\min}(t)$ on transactions. If the market is under-bonding relative to demand, batches with higher $\Sigma_{\text{bond}}(b)$ capture a larger share of the transaction flow whose rewards are gated by $K_{\min}$, increasing the realized return to bonding and attracting additional bonding. If the market is over-bonding, the reward rate per bonded KOR on successful flow falls and bonding decreases.

4.2.5 Expected Yields

The yield decreases as more stakers join, creating natural equilibrium where entry occurs until yield matches required return.

Example (with suggested parameters, $\sigma_{\min} = 10\text{M KOR}$, $\chi_{\text{consensus}} = 0.10$, and assuming bonded stake usage is proportional to total stake and near-continuous):

Stakers	Total Stake	Per-Staker Emissions	Yield (emissions only)
10	100M KOR	1M KOR/year	10.0%
30	300M KOR	333K KOR/year	3.3%
50	500M KOR	200K KOR/year	2.0%

4.3 Slashing Conditions

Stakers are slashed for protocol violations. Slashing severity varies by offense type and participation level.

4.3.1 Equivocation (Fatal)

A staker equivocates by signing two batches containing conflicting transactions:

$$\begin{aligned} \text{Equivocation}(s, b_1, b_2) :\equiv\ & s \in b_1.\text{signers} \wedge s \in b_2.\text{signers} \\ & \wedge \exists t_1 \in b_1.\text{txs}, t_2 \in b_2.\text{txs} : \text{Conflict}(t_1, t_2) \end{aligned} \tag{48}$$

Equivocation is provable malice. Penalty: λ_{equiv} of stake.

$$\text{penalty}_{\text{equiv}}(s) = \lambda_{\text{equiv}} \cdot \sigma_s \tag{49}$$

4.3.2 Invalid Batch (Severe)

A batch containing an invalid transaction (malformed, internal double-spend, etc.):

$$\text{InvalidBatch}(b) :\equiv \exists t \in b.\text{txs} : \neg\ \text{ValidTransaction}(t) \tag{50}$$

Signers failed their validation duty. Penalty: λ_{invalid} of bonded stake.

$$\text{penalty}_{\text{invalid}}(s, b) = \lambda_{\text{invalid}} \cdot \sigma_{s,b} \tag{51}$$

4.3.3 Conflict Confirmation (Severe)

When a conflicting transaction t_B confirms on Bitcoin instead of the batched t_A, stakers who signed the batch containing t_A are slashed. The staker asserted that no conflict would confirm; the assertion was false.

$$\text{ConflictConfirmed}(b, t_B) :\equiv \exists t_A \in b.\text{txs} : \text{Conflict}(t_A, t_B) \wedge \text{BitcoinConfirmed}(t_B) \tag{52}$$

Penalty: $\lambda_{\text{conflict}}$ of bonded stake.

$$\text{penalty}_{\text{conflict}}(s, b) = \lambda_{\text{conflict}} \cdot \sigma_{s,b} \tag{53}$$

User Griefing Defense: If a user submits a transaction to stakers, receives batching, then broadcasts a conflicting transaction to Bitcoin (griefing attack), stakers can submit proof that the user signed both conflicting transactions. In this case, stakers are exonerated and the user forfeits their Kontor execution fee (KOR). This proof consists of:

- The signed batch containing t_A
- The Bitcoin-confirmed conflicting transaction t_B
- Proof that both transactions were signed by the same key (the user)

4.3.4 Batch Expiry (Graduated)

When transactions in a batch fail to confirm within the expiry window W, signers are penalized. The penalty scales inversely with signature participation—fewer signers means each bears more responsibility.

Let $\varphi \stackrel{\text{def}}{=} \Sigma_{\text{signers}} \frac{b}{\Sigma}$ be the fraction of total stake that signed batch b, where $\varphi \in \left[\frac{2}{3}, 1\right]$.

Let $n_{\text{expired}}(b)$ be the number of transactions in batch b that expired without Bitcoin confirmation. Total batch penalty:

$$P_{\text{batch}} \stackrel{\text{def}}{=} \sigma_{\text{expiry}} \cdot n_{\text{expired}}(b) \cdot \left(\frac{1}{\varphi}\right) \tag{54}$$

Per-staker penalty (distributed among signers):

$$\text{penalty}_{\text{expiry}}(s, b) \stackrel{\text{def}}{=} \min\left(P_{\text{batch}} \cdot \left(\frac{\sigma_{s,b}}{\Sigma_{\text{bond}}}(b)\right), \lambda_{\text{cap}} \cdot \sigma_{s,b}\right) \tag{55}$$

where λ_{cap} caps any single batch's damage to a fraction of bonded stake.

The penalty roughly doubles when going from full participation to minimum quorum:

Signature %	**Total Penalty**
100%	σ_{expiry}
90%	$1.11 \cdot \sigma_{\text{expiry}}$
80%	$1.25 \cdot \sigma_{\text{expiry}}$
67%	$1.49 \cdot \sigma_{\text{expiry}}$

4.3.5 Slash Distribution

Slashed funds are split between burning and rewarding the evidence submitter:

$$\begin{aligned} \text{burn} &= \beta_{\text{slash}} \cdot \text{amount} \\ \text{reward} &= (1 - \beta_{\text{slash}}) \cdot \text{amount} \end{aligned} \tag{56}$$

Slashing and bond exposure: Slashing reduces a staker's stake. If slashing causes $\text{BondExposure}(s, \text{state}) > \sigma_s$, then any further batches signed by s will fail the bond exposure invariant until active batches resolve and bond exposure falls below remaining stake. Slashing for non-fatal batch failures is charged against the staker's pooled stake; realized slashing is capped by the staker's remaining stake. Equivocation (which slashes total stake) immediately removes the staker from the active set and marks their participation in other pending batches as zero-weight for future slashing purposes (though their signatures remain valid for quorum calculation on already-signed batches).

4.3.6 Liveness Failures

Repeated failure to participate in consensus rounds results in:

(a) Reduced reward share (proportional to missed rounds)
(b) Eventual ejection from staker set if below participation threshold
(c) Stake returned (not slashed) since liveness failures may be due to network issues

4.3.7 Fraud Proofs

Any user can submit fraud proofs to trigger slashing:

Equivocation proof: Two signed batches containing conflicting transactions. Self-contained cryptographic evidence.

Invalid batch proof: A signed batch containing a demonstrably invalid transaction.

Conflict proof: A signed batch plus Bitcoin proof that a conflicting transaction confirmed (unless griefing proof exonerates stakers).

Griefing proof: A signed batch containing t_A plus Bitcoin-confirmed t_B where $\text{Conflict}(t_A, t_B)$ and both transactions were signed by the same user key. Exonerates stakers from conflict slashing.

Expiry proof: A signed batch plus Bitcoin proof that transactions did not confirm within window W.

Note: Non-publication is not slashable (cannot prove a negative).

4.4 Economic Security

4.4.1 Attack Costs

Corrupting $\frac{1}{3}$ of stake enables equivocation attacks (signing conflicting batches). The attack is profitable only if the value extracted exceeds the stake destroyed:

$$\text{AttackCost} \geq \frac{1}{3} \cdot \Sigma \tag{57}$$

Bonds and security: The per-batch bond $\sigma_{s,b}$ does not change BFT safety (votes remain stake-weighted by σ_s), but it does change the strength of the economic guarantee a transaction can purchase. Transactions specify a minimum total bond requirement $K_{\min}(t)$, and ordering rewards for t are payment-gated: if $\Sigma_{\text{bond}}(b) < K_{\min}(t)$, stakers are not paid for t. This produces a market for economic finality without changing BFT assumptions.

Deliberately bad batches: A quorum can always choose to sign batches that are "bad" in the economic sense (e.g., including transactions that are unlikely to confirm before expiry). Under the assumed honest supermajority, this is bounded by slashing and by the fee-escrow

rule: stakers are paid only when transactions become Batch-Confirmed, while expiry and conflict events trigger penalties. If a coalition controlling $\geq \frac{2}{3}$ of total stake is willing to burn money to harm users, no purely economic mechanism can prevent service degradation; the protocol's guarantee in that case is graceful fallback to Bitcoin and objective evidence of what was signed.

Example (illustrative KOR prices):

Total Stake Σ	**Corruption Cost**	**At $0.20/KOR**
100M KOR	33M KOR	$6.6M
300M KOR	100M KOR	$20M
500M KOR	167M KOR	$33M

4.4.2 Rational Staker Behavior

A rational staker maximizes expected profit:

$$\mathbb{E}[\text{profit}(s)] = r_{\text{staker}}(s) - \mathbb{E}[\text{penalties}(s)] - \sigma_s \cdot \rho \tag{58}$$

where ρ is the opportunity cost of capital.

Honest behavior dominates because:

(a) Emissions and fees accrue to honest stakers
(b) Equivocation destroys entire stake (penalty λ_{equiv})
(c) Invalid batches and conflicts cost λ_{invalid}, $\lambda_{\text{conflict}}$ of stake
(d) Even expiry penalties exceed any fee savings from careless validation

4.4.3 Incentive Alignment

Staker Action	**Outcome**	**Incentive Effect**
Sign valid batch that confirms	Paid emissions + fees	Rewarded
Sign valid batch, conflict confirms	Rollback; $\lambda_{\text{conflict}}$ slashed	Deterred
Sign batch that expires	Penalized (graduated)	Deterred
Sign invalid batch	λ_{invalid} stake slashed	Strongly deterred
Equivocate	λ_{equiv} stake slashed	Catastrophically deterred
Censor transactions	Forgo fees	Opportunity cost

4.4.4 Miner Participation

Bitcoin miners have a natural economic incentive to participate as consensus stakers. A miner who also stakes KOR earns both mining revenue (block rewards plus Bitcoin transaction fees) and staking revenue (KOR emissions and execution fees). More importantly, miner-stakers can provide stronger guarantees: when a miner signs a batch, they are implicitly committing to include those transactions in blocks they mine. This reduces the probability of conflict between batched and mined transactions, since the same economic actors control both. The result is a virtuous cycle: miner participation increases staking finality confidence, which increases Kontor adoption, which increases Kontor execution fee revenue, which further incentivizes miner participation.

5 Security Analysis

5.1 Security Properties

BFT safety is inherited from the underlying consensus protocol. We establish the following core properties:

(a) **Ordering Determinism:** Same batches → same execution order
(b) **Finality Equivalence:** WasBatched ∧ Bitcoin-Confirmed ⇔ Batch-Confirmed

The forward direction of Finality Equivalence (⇒) follows from the definition of Batch-Confirmed. The reverse direction (⇐) holds because if the same transaction is Bitcoin-Confirmed, no conflicting transaction can be Bitcoin-Confirmed (Bitcoin prevents double-spends), so the transaction cannot be rolled back without a Bitcoin reorganization.

5.1.1 Safety

This is a standard BFT quorum intersection argument, inherited from the underlying consensus protocol.

Theorem (No Conflicting Batched Transactions): If honest stakers hold $> \frac{2}{3}$ of total stake, then for all valid batches b_1, b_2:

$$\forall t_1 \in b_1.\text{txs}, t_2 \in b_2.\text{txs} : \text{Conflict}(t_1, t_2) \Rightarrow (b_1 = b_2 \wedge t_1 = t_2) \tag{59}$$

Proof sketch:
(a) Quorum requires $\frac{2}{3}\Sigma$ stake.
(b) Any two quorums overlap by $\geq \frac{1}{3}\Sigma$ stake.
(c) Overlap contains at least one honest staker.
(d) Honest stakers refuse to sign batches containing conflicts with previously-signed batches.
(e) Therefore, conflicting transactions cannot appear in two valid batches. □

5.1.2 Determinism

Theorem (State Determinism): All honest nodes compute identical Kontor state from identical Bitcoin state and batch set:

$$\forall s_1, s_2 : (s_1.\text{bitcoin} = s_2.\text{bitcoin} \wedge s_1.\text{batches} = s_2.\text{batches}) \Rightarrow s_1.\text{kontor} = s_2.\text{kontor} \tag{60}$$

Proof: Execution order is deterministic (sorted by position), conflict resolution is deterministic (Bitcoin authoritative), and state transitions are deterministic functions. □

5.1.3 Rollback Detectability

Theorem: If a batched transaction is rolled back, there exists Bitcoin evidence (a conflicting transaction in a confirmed block). Rollback cascades to all subsequent transactions.

Proof: Rollback occurs only when a conflicting transaction confirms on Bitcoin. Bitcoin confirmations are publicly observable. Cascading follows from sequential execution semantics. □

5.1.4 Accountability

Theorem (Attributable Faults): Any safety violation is attributable to $\geq \frac{1}{3}\Sigma$ stake.

Proof sketch:
(a) Safety violation requires two conflicting batches with valid quorum signatures.
(b) Quorums overlap by $\geq \frac{1}{3}\Sigma$.
(c) Stakers in overlap signed both batches.
(d) Their identities are recorded in both `batch.signers`. □

5.1.5 Censorship Resistance

Property (Censorship Cost): A staker who censors transaction t forgoes expected fee share $(1 - \beta_{\text{fee}}) \cdot \delta_t \cdot \left(\frac{\sigma_{s,b}}{\Sigma_{\text{bond}}}(b)\right)$ on any batch b that would have included t, weighted by the probability p that t would have become Batch-Confirmed.

Property (Censorship Fallback): Users can bypass stakers entirely. Transaction confirms on Bitcoin and is appended at block-end.

5.1.6 Liveness

Property (Graceful Degradation): If staker liveness fails, users can still transact via Bitcoin directly. Transactions confirm and append at block-end without optimistic ordering.

5.2 Cryptographic Proofs

We prove that any safety violation can be attributed to specific stakers whose stake exceeds the Byzantine threshold.

5.2.1 Security Model

Let $\Sigma = (\text{KeyGen}, \text{Sign}, \text{Verify})$ be an EUF-CMA secure signature scheme.

Definition (EUF-CMA Security): A signature scheme is EUF-CMA secure if for any PPT adversary $\mathcal{A}$:

$$\text{Adv}_{\Sigma}^{\text{EUF-CMA}}(\mathcal{A}) := \Pr\left[\text{EUF-CMA}_{\Sigma}^{\mathcal{A}} = 1\right] \leq \text{negl}(\lambda) \tag{61}$$

where the EUF-CMA game allows $\mathcal{A}$ to query a signing oracle and must output a valid signature on a message not queried.

5.2.2 Accountability Theorem

Theorem (Accountable Safety): Let $\mathcal{P}$ be the Kontor Optimistic Consensus protocol using signature scheme Σ. If Σ is EUF-CMA secure, then any safety violation is attributable to a set of stakers holding $\geq \frac{1}{3}$ of total stake.

Proof:

Suppose a safety violation occurs: there exist two valid batches b_1, b_2 containing transactions $t_1 \in b_1.\text{txs}$ and $t_2 \in b_2.\text{txs}$ with $\text{Conflict}(t_1, t_2)$.

Step 1: Quorum Intersection.

Since both batches are valid:

$$\text{ValidQuorum}(b_1.\text{signers}, S) \wedge \text{ValidQuorum}(b_2.\text{signers}, S) \tag{62}$$

By the Quorum Intersection Lemma:

$$\sum_{s \in b_1.\text{signers} \cap b_2.\text{signers}} S.\text{stakers}[s] \geq \frac{1}{3} \cdot \text{TotalStake}(S) \tag{63}$$

Step 2: Signature Evidence.

For each staker $s \in b_1.\text{signers} \cap b_2.\text{signers}$:

- There exists σ_1 such that $\text{Verify}(\sigma_1, \text{BatchDigest}(b_1), \text{pk}_s) = 1$
- There exists σ_2 such that $\text{Verify}(\sigma_2, \text{BatchDigest}(b_2), \text{pk}_s) = 1$

Since $b_1 \neq b_2$ (they contain conflicting transactions), we have $\text{BatchDigest}(b_1) \neq \text{BatchDigest}(b_2)$.

Step 3: Attribution.

Define the guilty set:

$$G := \{s \in b_1.\text{signers} \cap b_2.\text{signers} : \text{ValidSignature}(\sigma_1^s, s, b_1) \wedge \text{ValidSignature}(\sigma_2^s, s, b_2)\} \tag{64}$$

By Step 1, $\sum_{s \in G} S.\text{stakers}[s] \geq \frac{1}{3} \cdot \text{TotalStake}(S)$.

Step 4: Non-Repudiation.

Suppose staker $s \in G$ claims innocence (did not sign one of the batches). Without loss of generality, suppose s claims they did not sign b_1.

Construct an EUF-CMA adversary $\mathcal{B}$:

- $\mathcal{B}$ receives pk_s as challenge public key
- $\mathcal{B}$ simulates the protocol, using the signing oracle for s's signatures
- When the safety violation occurs, $\mathcal{B}$ outputs $(\sigma_1^s, \text{BatchDigest}(b_1))$

If s truly did not sign b_1, then $\text{BatchDigest}(b_1)$ was never queried to the oracle, so $(\sigma_1^s, \text{BatchDigest}(b_1))$ is a valid EUF-CMA forgery.

This contradicts EUF-CMA security of Σ. Therefore, s must have signed both batches. □

5.2.3 Slashing Correctness

Corollary (Slashing Correctness): The slashing mechanism is sound: an honest staker is never slashed, and any slashed staker is guilty of a protocol violation.

Proof:

Soundness (no false positives): An honest staker only signs batches that do not conflict with previously-signed batches. By the honest staker rule, if s is honest and $b \in \text{signed}_s$, then for all $b' \in \text{signed}_s$: $\forall t \in b.\text{txs}, t' \in b'.\text{txs} : \neg\ \text{Conflict}(t, t')$.

Slashing requires equivocation evidence: two batches with conflicting transactions both signed by s. This cannot occur for honest s.

Completeness (guilty stakers slashable): By the Accountability Theorem, any safety violation produces cryptographic evidence (two signatures by the same staker on conflicting batches). This evidence constitutes valid slashing evidence. □

5.3 Game-Theoretic Analysis

5.3.1 Inclusion Incentive

Setting: Stakers collectively decide which transactions to include in batches.

Staker payoff for including transaction t:

$$\pi_{s(t)} = \begin{cases} (1 - \beta_{\text{fee}}) \cdot \delta_t \cdot \left(\frac{\sigma_s}{\Sigma_{\text{signers}}}\right) & \text{if } t \text{ confirms before expiry} \\ 0 & \text{if } t \text{ expires or is rolled back} \end{cases} \tag{65}$$

Theorem (Inclusion Incentive): A utility-maximizing staker includes all valid transactions.

Proof:

Let t be a valid transaction with fee δ_t and confirmation probability $p > 0$.

Expected payoff from including t:

$$\mathbb{E}\left[\pi_{s(t)} \mid \text{include}\right] = p \cdot (1 - \beta_{\text{fee}}) \cdot \delta_t \cdot \left(\frac{\sigma_s}{\Sigma_{\text{signers}}}\right) \tag{66}$$

Expected payoff from excluding t:

$$\mathbb{E}\left[\pi_{s(t)} \mid \text{exclude}\right] = 0 \tag{67}$$

Since $p > 0$ and $\delta_t > 0$, inclusion strictly dominates exclusion. □

5.3.2 Censorship Cost

Censorship has a quantifiable cost: foregone fees. For a coalition C censoring transaction t:

$$\text{CensorshipCost}(C, t) = (1 - \beta_{\text{fee}}) \cdot \delta_t \cdot \left(\frac{\Sigma_C}{\Sigma_{\text{signers}}}\right) \tag{68}$$

where Σ_C is the total stake of the censoring coalition.

Case 1: C does not form a quorum. Then t is included by honest stakers anyway, and C merely forgoes their share.

Case 2: C forms a quorum ($\geq \frac{2}{3}$ stake). The user bypasses stakers and broadcasts to Bitcoin. Transaction confirms at block-end; C receives nothing.

Censorship is profitable if and only if the external bribe $B > \text{CensorshipCost}(C, t)$. The protocol does not prevent censorship—it makes the cost transparent and quantifiable.

5.4 Lean 4 Formalization

The protocol's fundamental claim: Bitcoin provides finality, batches provide ordering—these are orthogonal. The following Lean 4 code provides machine-checked proofs of three properties:

(a) **Ordering determinism:** Same batches and unbatched transaction lists produce the same execution order, regardless of L1 state.
(b) **Finality inheritance:** Kontor-final implies L1-final—no transaction is considered final without Bitcoin confirmation.
(c) **Conflict handling:** When a conflicting transaction (RBF, double-spend) confirms on L1, the batched transaction is invalidated.

Ordering is static—determined by batch structure and block positions. *Validity* is dynamic—a transaction's position is fixed, but whether it executes depends on L1 state (expiry, conflicts). Together these prove: (1) we inherit Bitcoin's finality, (2) ordering is deterministic, and (3) conflicts are handled correctly.

```
import Mathlib.Data.Finset.Basic
import Mathlib.Data.List.Basic

namespace KontorConsensus

/-! ## Core Types -/

abbrev TxId := Nat
abbrev BlockHeight := Nat
abbrev UTXO := Nat

/-- A transaction with inputs -/
structure Transaction where
  id : TxId
  inputs : Finset UTXO
  deriving DecidableEq

/-- A batch is an ordered sequence of transactions with an expiry -/
structure Batch where
  id : Nat
  txs : List Transaction
  expiry : BlockHeight
  deriving DecidableEq

/-- An unbatched transaction confirmed on L1, with its block position -/
structure UnbatchedTx where
  tx : Transaction
  block_height : BlockHeight
  block_index : Nat
  deriving DecidableEq

/-- Protocol state -/
```

```
structure State where
  l1_height : BlockHeight
  l1_confirmed : Finset TxId
  batches : List Batch
  unbatched : List UnbatchedTx

/-! ## Ordering

Ordering is static: determined entirely by batch structure and block
positions.
Execution order: batched transactions first (by batch), then unbatched at
block-end.
-/

/-- Position of a batched transaction: (batch_id, index_in_batch) -/
def batchPosition (batches : List Batch) (tx : Transaction) : Option (Nat ×
Nat) :=
  batches.enum.findSome? fun (_, batch) =>
    batch.txs.enum.findSome? fun (tx_idx, t) =>
      if t.id = tx.id then some (batch.id, tx_idx) else none

/-- Sort unbatched transactions by (block_height, block_index) -/
def sortUnbatched (unbatched : List UnbatchedTx) : List UnbatchedTx :=
  unbatched.mergeSort fun a b =>
    a.block_height < b.block_height ||
    (a.block_height = b.block_height && a.block_index ≤ b.block_index)

/-- Execution order: batched first, then unbatched sorted by block position -/
def executionOrder (s : State) : List Transaction :=
  let batched := s.batches.bind (·.txs)
  let unbatchedSorted := (sortUnbatched s.unbatched).map (·.tx)
  batched ++ unbatchedSorted

/-! ## Validity

Validity is dynamic: depends on L1 state (expiry, conflicts).
A transaction's position is fixed, but whether it executes depends on L1.
-/

/-- Conflict relation: transactions share at least one input (RBF, double-
spend) -/
def conflict (tx1 tx2 : Transaction) : Prop :=
  tx1.id ≠ tx2.id ∧ (tx1.inputs ∩ tx2.inputs).Nonempty

/-- A batch is expired if current height exceeds expiry -/
def batchExpired (s : State) (b : Batch) : Prop :=
  s.l1_height > b.expiry

/-- A batched transaction is conflicted if a different tx spending the same
inputs confirmed on L1 -/
def isConflicted (s : State) (tx : Transaction) : Prop :=
  ∃ conflicting_tx : Transaction,
    conflicting_tx.id ∈ s.l1_confirmed ∧
    conflicting_tx.id ≠ tx.id ∧
    conflict tx conflicting_tx

/-- A batched transaction is valid if its batch hasn't expired and it's not
conflicted -/
def batchedTxValid (s : State) (tx : Transaction) : Prop :=
  ∃ b ∈ s.batches, tx ∈ b.txs ∧ ¬batchExpired s b ∧ ¬isConflicted s tx
```

```
/-- Transactions at or after a rolled-back position must also roll back -/
def cascadeRollback (batches : List Batch) (rollback_pos : Nat × Nat) (tx :
Transaction) : Prop :=
  match batchPosition batches tx with
  | some pos => pos.1 > rollback_pos.1 ∨ (pos.1 = rollback_pos.1 ∧ pos.2 ≥
rollback_pos.2)
  | none => False

/-! ## Property 1: Ordering Determinism

Ordering is static—same batches and unbatched lists yield the same order.
L1 state affects validity, not ordering.
-/

/-- Ordering depends only on batches and unbatched list, not L1 state -/
theorem ordering_determinism (s1 s2 : State)
    (h_batches : s1.batches = s2.batches)
    (h_unbatched : s1.unbatched = s2.unbatched) :
    executionOrder s1 = executionOrder s2 := by
  simp only [executionOrder, h_batches, h_unbatched]

/-- Batch position depends only on batches -/
theorem batch_position_determinism (s1 s2 : State)
    (h_batches : s1.batches = s2.batches)
    (tx : Transaction) :
    batchPosition s1.batches tx = batchPosition s2.batches tx := by
  simp only [h_batches]

/-! ## Property 2: Finality Inheritance

Finality comes from L1, not from batching. Kontor-final implies L1-final.
-/

/-- L1 finality: transaction confirmed on Bitcoin -/
def l1Final (s : State) (tx : Transaction) : Prop :=
  tx.id ∈ s.l1_confirmed

/-- Kontor finality: L1 confirmation of a valid batched tx, or unbatched
confirmation -/
def kontorFinal (s : State) (tx : Transaction) : Prop :=
  l1Final s tx ∧ (batchedTxValid s tx ∨ ∃ utx ∈ s.unbatched, utx.tx = tx)

/-- Finality inheritance: Kontor-final implies L1-final -/
theorem finality_requires_l1
    (s : State) (tx : Transaction)
    (h : kontorFinal s tx) :
    l1Final s tx :=
  h.1

/-- Contrapositive: no L1 confirmation means no Kontor finality -/
theorem no_l1_no_kontor_final
    (s : State) (tx : Transaction)
    (h_not_l1 : ¬l1Final s tx) :
    ¬kontorFinal s tx := by
  intro h_kontor
  exact h_not_l1 h_kontor.1

/-! ## Property 3: Conflict Handling

When a conflicting transaction confirms on L1, the batched transaction is
invalidated.
```

```
This models RBF and double-spend scenarios.
-/

/-- If a conflict confirms, the batched tx becomes invalid -/
theorem conflict_invalidates
    (s : State) (tx : Transaction) (conflict_tx : Transaction)
    (h_conflict : conflict tx conflict_tx)
    (h_confirmed : conflict_tx.id ∈ s.l1_confirmed) :
    isConflicted s tx := by
  exact ⟨conflict_tx, h_confirmed, h_conflict.1.symm, h_conflict⟩

/-- Conflicted transactions are not valid -/
theorem conflicted_not_valid
    (s : State) (tx : Transaction)
    (h_conflicted : isConflicted s tx) :
    ¬batchedTxValid s tx := by
  intro ⟨_, _, _, _, h_not_conflicted⟩
  exact h_not_conflicted h_conflicted

end KontorConsensus
```

6. Bibliography

[1] Adam Krellenstein, Wilfred Denton, and Ouziel Slama, "Kontor: A New Bitcoin Metaprotocol for Smart Contracts and File Persistence," 2025. [Online]. Available: https://docs.kontor.network/docs/resources/whitepaper

[2] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham, "HotStuff: BFT Consensus with Linearity and Responsiveness," 2019.

[3] Satoshi Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[4] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer, "Consensus in the Presence of Partial Synchrony," 1988.

[5] Miguel Castro and Barbara Liskov, "Practical Byzantine Fault Tolerance," 1999.

[6] Ethan Buchman, Jae Kwon, and Zarko Milosevic, "The latest gossip on BFT consensus," 2018. [Online]. Available: https://arxiv.org/abs/1807.04938

[7] Adam Krellenstein, "Kontor BTC Yield," 2025. [Online]. Available: https://docs.kontor.network/docs/resources/btc-yield