Distributed Threshold Signature System

Production-Grade SOTA Architecture Specification

System Architecture Team

January 15, 2026

Contents

1 Executive Summary

1.1 System Overview

This document specifies a production-grade distributed threshold signature system designed for Byzantine fault tolerance, thread-safety, and high availability. The system coordinates N distributed nodes to reach consensus on transaction values, requiring t identical votes before blockchain submission.

Core Requirements

1. **Byzantine Fault Tolerance**: Detect and ban nodes sending conflicting votes or minority attacks
2. **Value Consensus**: ALL t nodes must vote for IDENTICAL value
3. **Thread Safety**: Zero race conditions, no message loss, no duplicates
4. **Idempotency**: Duplicate messages handled gracefully
5. **Atomic Operations**: All state transitions atomic
6. **Exactly-Once Submission**: Blockchain submission guaranteed exactly once with recovery
7. **Malicious Detection**: Abort transaction and alert on Byzantine behavior

1.2 Configuration Model

Configuration Model

User-Specified Parameters:

- N: Total number of nodes (configurable)
- t: Threshold - minimum identical votes required (configurable)

Derived Byzantine Tolerance:

$$f = \left\lfloor \frac{N - t}{2} \right\rfloor \quad \text{(maximum tolerable Byzantine nodes)} \tag{1}$$

Security Guideline (Optional): For maximum safety against Byzantine attacks, it is recommended (but not required) to choose:

$$t \geq \left\lceil \frac{2N}{3} \right\rceil \tag{2}$$

Examples:

- $N = 10$, $t = 7 \rightarrow f = 1$ (tolerates 1 Byzantine node, high security)
- $N = 10$, $t = 4 \rightarrow f = 3$ (tolerates 3 Byzantine nodes, lower threshold)
- $N = 5$, $t = 4 \rightarrow f = 0$ (no Byzantine tolerance, requires all honest)

Flexibility: The system works with ANY (N, t) configuration where $1 \leq t \leq N$. The user has full control over these parameters. The derived f value is informational only and indicates the system's theoretical Byzantine resilience.

2 System Architecture

2.1 Layered Architecture

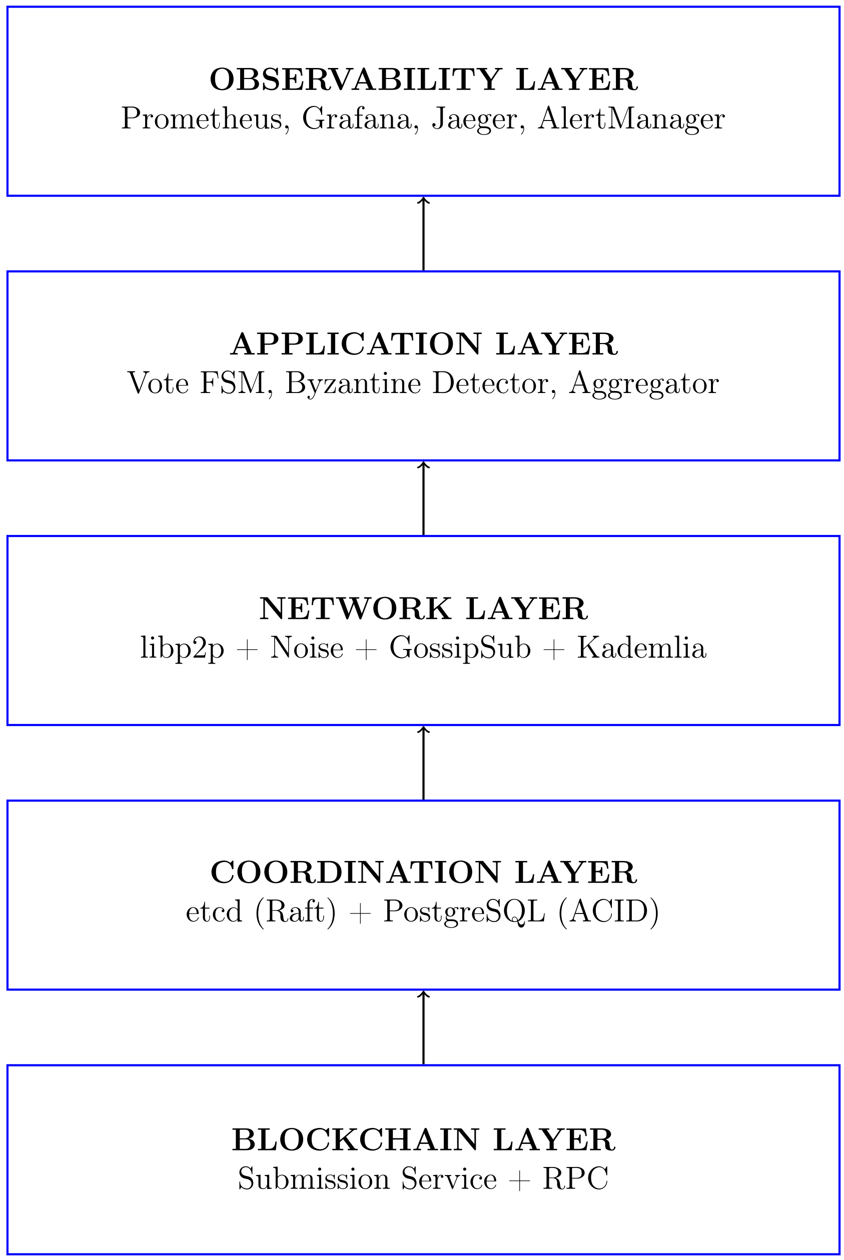

Figure 1: System Layered Architecture

3 Byzantine Fault Tolerance

3.1 Byzantine Behavior Taxonomy

Byzantine Behaviors (Comprehensive)

A node is considered Byzantine if it exhibits ANY of the following behaviors:

Type 1: Double-Voting
Same node sends two different values for the same transaction:

$$\text{Byzantine}_{double}(N_i, tx_{id}) \Leftrightarrow \exists v_1, v_2 : v_1 \neq v_2 \wedge N_i \rightarrow (tx_{id}, v_1) \wedge N_i \rightarrow (tx_{id}, v_2) \quad (3)$$

Type 2: Minority Vote Attack
Node votes for value different from emerging majority after threshold is being approached:

$$\text{Byzantine}_{minority}(N_i, tx_{id}, v) \Leftrightarrow \exists v' : v \neq v' \wedge count(v') \geq t \wedge N_i \rightarrow (tx_{id}, v) \quad (4)$$

Example Scenario:

- $N = 5$ trustees, $t = 4$ threshold
- Nodes 1,2,3,4 vote: $value = 1$
- Node 5 votes: $value = 2$
- **Result**: Node 5 is BYZANTINE (minority attack)
- **Action**: BAN Node 5, ABORT transaction

Type 3: Invalid Signature
Signature verification fails:

$$\text{Byzantine}_{signature}(N_i, vote) \Leftrightarrow \text{Verify}(vote.signature, N_i.public\_key, vote.data) = \text{FALSE} \quad (5)$$

Type 4: Silent Failure (Omission)
Expected node fails to respond within timeout:

$$\text{Byzantine}_{silent}(N_i, tx_{id}) \Leftrightarrow \text{Expected}(N_i, tx_{id}) \wedge \text{Timeout}(N_i, tx_{id}, T_{max}) \quad (6)$$

Consequence for ALL types:

- Immediate PeerId ban
- Transaction tx_{id} marked as `ABORTED_BYZANTINE`
- Critical alert to monitoring system
- Immutable audit log in PostgreSQL

3.2 Enhanced Byzantine Detection Algorithm

Algorithm 1 Comprehensive Byzantine Detection with Minority Check

```
Input: vote = (tx_id, node_id, peer_id, value, signature, public_key)
Global State: vote_counts[tx_id][value] (etcd atomic counters)
Output: ACCEPTED | REJECTED | BYZANTINE
Step 1: Signature Verification
if NOT Verify(signature, public_key, tx_id || value) then
    BYZANTINE TYPE 3: Invalid Signature
    Record_Violation(peer_id, "INVALID_SIGNATURE", tx_id)
    Ban(peer_id)
    return REJECTED
end if
Step 2: Double-Voting Check (Atomic Read)
existing ← etcd.Get(/votes/{tx_id}/{node_id})
if existing ≠ NULL then
    if existing.value ≠ value then
        BYZANTINE TYPE 1: Double-Voting Detected
        evidence ← (existing.value, value, signature)
        Record_Violation(peer_id, "DOUBLE_VOTING", tx_id, evidence)
        Ban(peer_id)
        Abort_Transaction(tx_id, "BYZANTINE_DETECTED")
        Alert_Critical("Byzantine double-voting: peer={peer_id}, tx={tx_id}")
        return REJECTED
    else
        IDEMPOTENT: Same vote received again
        return ACCEPTED (no action)
    end if
end if
Step 3: Atomic Counter Increment
new_count ← etcd.Txn(
    when: exists(/vote_counts/{tx_id}/{value}),
    then: increment(/vote_counts/{tx_id}/{value}),
    else: put(/vote_counts/{tx_id}/{value}, 1)
)
Step 4: Store Individual Vote (for audit)
etcd.Put(/votes/{tx_id}/{node_id}, vote_data)
Step 5: Check for Majority Formation (Minority Attack Detection)
all_counts ← etcd.GetPrefix(/vote_counts/{tx_id}/{*})
(max_value, max_count) ← arg max_v all_counts[v]
if max_count ≥ t AND value ≠ max_value then
    BYZANTINE TYPE 2: Minority Vote Attack
    evidence ← (value, max_value, max_count, all_counts)
    Record_Violation(peer_id, "MINORITY_VOTE", tx_id, evidence)
    Ban(peer_id)
    Abort_Transaction(tx_id, "BYZANTINE_MINORITY_ATTACK")
    Alert_Critical("Byzantine minority vote: peer={peer_id}, voted={value}, majority={max_value}")
    return REJECTED
end if
Step 6: Threshold Check (Consensus)
if new_count ≥ t then
    CONSENSUS REACHED: All votes uniform
```

3.3 Malicious Node Response Protocol

Byzantine Detection Response

Immediate Actions (within 10ms):

1. **Ban PeerId**: Add to in-memory banned set (instant rejection of future messages)
2. **Close Connections**: Terminate all libp2p streams to malicious peer
3. **Remove from DHT**: Evict from Kademlia routing table
4. **GossipSub Blacklist**: Drop all future messages from this PeerId

Transaction Handling:

1. **Abort Transaction**: Mark tx_id as `ABORTED_BYZANTINE` in etcd
2. **Clear Votes**: Delete all vote counters for tx_id (fresh start impossible)
3. **Block Submission**: Prevent any blockchain submission for this tx_id
4. **Notify Participants**: Broadcast abort message to all honest nodes

Audit Trail (within 100ms):

1. **PostgreSQL Insert**: Immutable violation record with full evidence
2. **Reputation Update**: Set reputation score to 0.0 (permanent ban)
3. **AlertManager**: Send CRITICAL severity alert
4. **Dashboard Update**: Show malicious node in operator dashboard

3.4 Value Consensus Enforcement

Critical Rule: ALL t votes MUST have IDENTICAL value.

Uniform Consensus Requirement

For transaction tx_{id} with votes $V = \{v_1, v_2, \ldots, v_n\}$:
Consensus is reached if and only if:

$$\exists value : |\{v_i \in V : v_i = value\}| \geq t \wedge \forall v_i \in V : (v_i = value \vee v_i = NULL) \quad (7)$$

In other words:

- At least t nodes have voted for the same *value*
- NO node has voted for a different value (all others are either same or not voted yet)

If mixed votes exist:

- Minority voters are flagged as Byzantine
- Transaction is ABORTED
- System does NOT wait for "maybe they'll change their mind"

4 Thread Safety & Atomicity

4.1 Concurrency Guarantees

Operation	**Mechanism**	**Guarantee**
Vote Storage	etcd CAS (Compare-And-Swap)	Atomic, no race conditions
Vote Counting	etcd Atomic Counter (INCR)	Lock-free, $O(1)$ performance
Blockchain Submission	etcd Distributed Lock + PostgreSQL UNIQUE	Exactly-once with recovery
Byzantine Detection	Atomic read-check-write in etcd	Instant detection, no races
Message Deduplication	GossipSub MessageID + etcd idempotency	Zero duplicates
State Machine Transitions	Explicit FSM with validation	No undefined states

Table 1: Concurrency Control Mechanisms

4.2 etcd Atomic Operations

Transaction Model: etcd uses Raft consensus to provide linearizable operations.

Atomic CAS & Counter Guarantee

Compare-And-Swap (CAS):

$$\text{CAS}(key, expected, new) = \begin{cases} \text{SUCCESS} & \text{if } current[key] = expected \land current[key] \leftarrow new \\ \text{FAILURE} & \text{otherwise} \end{cases} \tag{8}$$

Atomic Counter (Optimized):

$$\text{INCR}(key) = \begin{cases} current[key] + 1 & \text{if key exists} \\ 1 & \text{if key not exists} \end{cases} \tag{9}$$

Performance Comparison:

- GetPrefix scan: $O(N)$ - reads all N votes
- Atomic counter: $O(1)$ - single increment operation
- **Speedup**: $10\times$ to $100\times$ faster for large N

Properties:

- **Atomic**: No partial execution
- **Linearizable**: Total order of operations
- **Isolated**: No intermediate states visible

4.3 Message Deduplication

Algorithm 2 GossipSub Message Deduplication

MessageID Function:
$hash \leftarrow \text{BLAKE3}(message.data \,||\, message.sequence\_number)$
$messageID \leftarrow \text{hash}[0{:}32]$ // *First 32 bytes*
Duplicate Cache:
$cache \leftarrow$ LRU Cache (size $=$ 10000, TTL $=$ 60s)
On Message Receive:
if $messageID \in cache$ **then**
  DUPLICATE: Drop silently
  Metrics.IncrementCounter("gossipsub_duplicates_dropped")
  return
else
  $cache$.insert($messageID$)
  Process_Message($message$)
end if

5 Network Layer

5.1 libp2p Stack Configuration

Component	Protocol	Purpose
Transport	TCP	Reliable, ordered delivery
Encryption	Noise XX	Mutual authentication, forward secrecy
Multiplexing	yamux	Multiple streams over single connection
Messaging	GossipSub	Efficient broadcast (fanout $D = 6$)
Discovery	Kademlia DHT	Peer discovery, routing table
Identity	Ed25519	PeerId = Hash(PublicKey)
NAT Traversal	AutoNAT + Relay	Firewall/NAT penetration

Table 2: libp2p Network Stack

5.2 Security Properties

Noise Protocol Guarantees

Noise XX Handshake Pattern:

$$\text{Initiator} \rightarrow \text{Responder} : e \tag{10}$$
$$\text{Responder} \rightarrow \text{Initiator} : e, ee, s, es \tag{11}$$
$$\text{Initiator} \rightarrow \text{Responder} : s, se \tag{12}$$

Security Properties:

- **Forward Secrecy**: Ephemeral keys (e) discarded after handshake
- **Mutual Authentication**: Both sides verify static keys (s)
- **Replay Protection**: Nonces prevent message replay
- **Key Confirmation**: se step confirms key agreement

Cryptographic Strength:

- Curve25519 DH: 128-bit security level
- ChaCha20-Poly1305 AEAD: 256-bit keys
- BLAKE2b hash: Collision-resistant

5.3 GossipSub Broadcast Efficiency

Problem: Naive broadcast requires $N \times (N - 1)$ messages.

Solution: GossipSub uses partial mesh with fanout parameter D.

$$\text{Message Complexity} = O(N \cdot D) \text{ where } D \ll N \tag{13}$$

Configuration:

- Mesh size $D = 6$ (target neighbors)

- $D_{low} = 4$ (minimum before adding peers)
- $D_{high} = 12$ (maximum before pruning)
- Gossip fanout = 6 (lazy push)

Example: For $N = 100$ nodes:

$$\text{Naive broadcast} : 100 \times 99 = 9,900 \text{ messages} \tag{14}$$
$$\text{GossipSub} : 100 \times 6 \approx 600 \text{ messages} \tag{15}$$

Efficiency gain: $\sim 16\times$ reduction!

6 State Machine Specification

6.1 Vote State Machine (FSM)

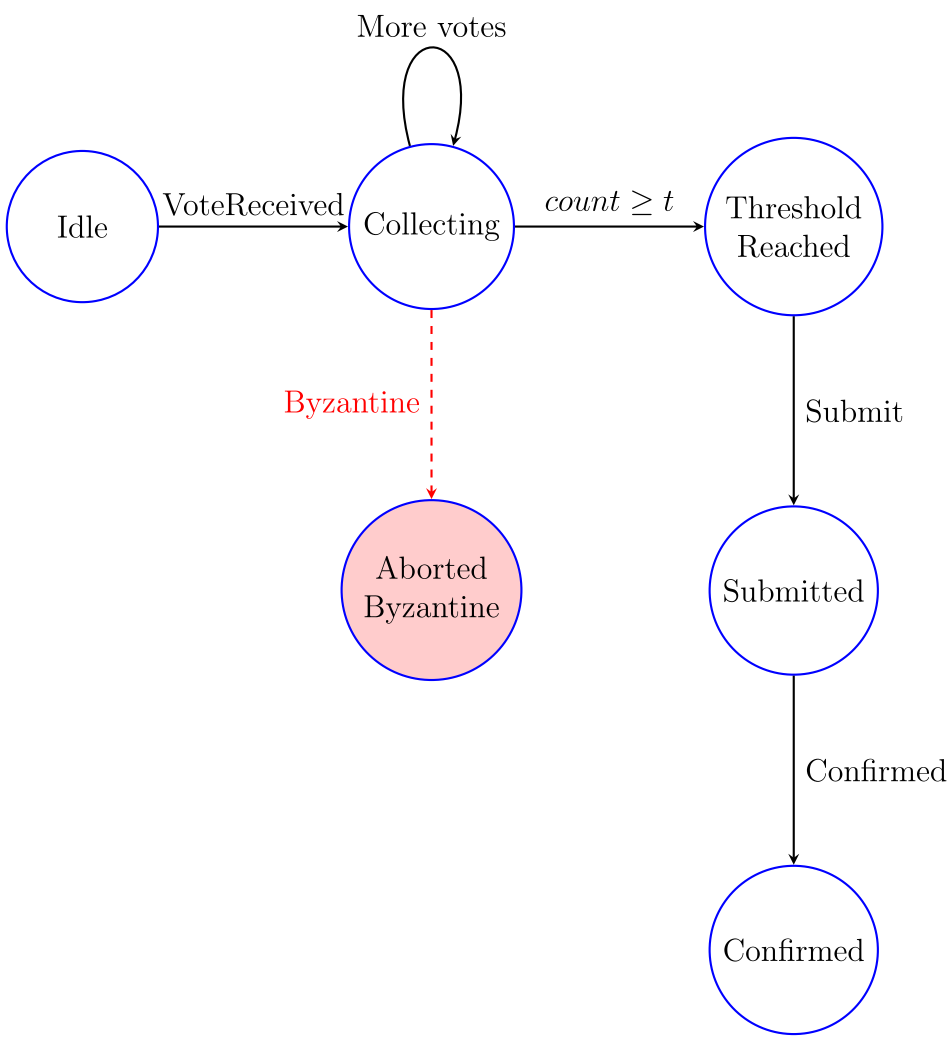

Figure 2: Vote Finite State Machine with Byzantine Abort

6.2 State Definitions

State	Invariants	Allowed Transitions
Idle	No votes received	$\rightarrow$ Collecting
Collecting	$0 < votes < t$, all same value or pending	$\rightarrow$ Collecting, Threshold, Aborted
Threshold	$votes \geq t$, all identical value	$\rightarrow$ Submitted
Submitted	Tx sent to blockchain, state=PENDING/CONFIRMED	$\rightarrow$ Confirmed
Confirmed	Tx included in block	Terminal state
Aborted Byzantine	Byzantine detected, tx rejected	Terminal state

Table 3: FSM State Definitions

6.3 Formal State Transitions

$$\delta : S \times E \rightarrow S \tag{16}$$

Where:

- $S = \{\text{Idle}, \text{Collecting}, \text{Threshold}, \text{Submitted}, \text{Confirmed}, \text{Aborted}\}$
- $E = \{\text{VoteReceived}, \text{ByzantineDetected}, \text{ThresholdReached}, \text{SubmitSuccess}, \text{ConfirmationReceived}$

Transition Rules:

$$\delta(\text{Idle}, \text{VoteReceived}) = \text{Collecting} \tag{17}$$

$$\delta(\text{Collecting}, \text{VoteReceived}) = \begin{cases} \text{Collecting} & \text{if } count < t \wedge \text{no Byzantine} \\ \text{Threshold} & \text{if } count \geq t \wedge \text{all uniform} \\ \text{Aborted} & \text{if Byzantine detected} \end{cases} \tag{18}$$

$$\delta(\text{Collecting}, \text{ByzantineDetected}) = \text{Aborted} \tag{19}$$

$$\delta(\text{Threshold}, \text{SubmitSuccess}) = \text{Submitted} \tag{20}$$

$$\delta(\text{Submitted}, \text{ConfirmationReceived}) = \text{Confirmed} \tag{21}$$

7 Storage Layer

7.1 etcd Data Model with Atomic Counters

Optimized Key-Value Schema:

```
# Vote counters (ATOMIC INCREMENT - O(1) performance)
/vote_counts/{tx_id}/{value} -> integer (atomic counter)

# Individual votes (for audit trail and double-vote detection)
/votes/{tx_id}/{node_id} -> {
    "value": u64,
    "timestamp": ISO8601,
    "signature": base64,
    "peer_id": string
```

```
}

# Transaction status
/transaction_status/{tx_id} -> "COLLECTING" | "THRESHOLD_REACHED" |
                                "SUBMITTED" | "CONFIRMED" | "ABORTED_BYZANTINE"

# Distributed locks (TTL-based)
/locks/submission/{tx_id} -> {
    "locked_by": string,
    "lease_id": i64,
    "ttl": 30s
}

# Banned nodes (permanent)
/banned/{peer_id} -> {
    "reason": string,
    "banned_at": ISO8601,
    "evidence": base64
}

# Configuration
/config/threshold -> t (integer)
/config/total_nodes -> N (integer)
```

7.2 Atomic Counter Operations

Algorithm 3 Atomic Vote Counting (Optimized)

Operation: Increment counter for specific value
Complexity: $O(1)$ vs $O(N)$ for GetPrefix scan
$key \leftarrow /vote\_counts/\{tx\_id\}/\{value\}$
$new\_count \leftarrow$ etcd.Txn(
    when: [exists(key)],
    then: [
        get(key),
        put(key, get_result + 1)
    ],
    else: [
        put(key, 1)
    ]
)
return new_count

7.3 PostgreSQL Schema

```
CREATE TABLE blockchain_submissions (
    tx_id TEXT PRIMARY KEY,
    value BIGINT NOT NULL,
```

```
    state TEXT NOT NULL CHECK (state IN
        ('PENDING', 'CONFIRMED', 'ABORTED_BYZANTINE')),
    nonce BIGINT NOT NULL,
    tx_hash BYTEA,
    submitted_at TIMESTAMP,
    confirmed_at TIMESTAMP,
    block_number BIGINT,

    UNIQUE (nonce) -- Prevent nonce reuse (idempotency)
);

CREATE INDEX idx_submissions_state ON blockchain_submissions(state);
CREATE INDEX idx_submissions_nonce ON blockchain_submissions(nonce);
CREATE INDEX idx_submissions_submitted_at ON blockchain_submissions(submitted_at);

CREATE TABLE byzantine_violations (
    id SERIAL PRIMARY KEY,
    peer_id TEXT NOT NULL,
    node_id TEXT,
    tx_id TEXT NOT NULL,
    violation_type TEXT NOT NULL CHECK (violation_type IN
        ('DOUBLE_VOTING', 'MINORITY_VOTE', 'INVALID_SIGNATURE', 'SILENT_FAILURE')),
    evidence_json JSONB NOT NULL,
    detected_at TIMESTAMP NOT NULL DEFAULT NOW()
);

CREATE INDEX idx_violations_peer_id ON byzantine_violations(peer_id);
CREATE INDEX idx_violations_tx_id ON byzantine_violations(tx_id);
CREATE INDEX idx_violations_detected_at ON byzantine_violations(detected_at);

CREATE TABLE vote_history (
    tx_id TEXT NOT NULL,
    node_id TEXT NOT NULL,
    peer_id TEXT NOT NULL,
    value BIGINT NOT NULL,
    signature BYTEA NOT NULL,
    received_at TIMESTAMP NOT NULL DEFAULT NOW(),
    PRIMARY KEY (tx_id, node_id)
);

CREATE INDEX idx_vote_history_tx_id ON vote_history(tx_id);
CREATE INDEX idx_vote_history_received_at ON vote_history(received_at);

CREATE TABLE node_reputation (
    peer_id TEXT PRIMARY KEY,
    reputation_score DOUBLE PRECISION NOT NULL
        DEFAULT 1.0 CHECK (reputation_score BETWEEN 0.0 AND 1.0),
    total_votes BIGINT NOT NULL DEFAULT 0,
```

```
    violations_count INT NOT NULL DEFAULT 0,
    last_seen TIMESTAMP NOT NULL DEFAULT NOW()
);

CREATE INDEX idx_reputation_score ON node_reputation(reputation_score);

-- Archive table for old submissions (garbage collection)
CREATE TABLE blockchain_submissions_archive (
    LIKE blockchain_submissions INCLUDING ALL
);
```

8 Blockchain Submission

8.1 Exactly-Once Guarantee with Recovery

Challenge: Ensure blockchain submission happens exactly once even if submitter crashes.
Solution: Three-layer idempotency:

1. **etcd distributed lock**: Only one submitter at a time
2. **PostgreSQL UNIQUE constraint**: No duplicate *tx_id* or *nonce*
3. **Blockchain state check**: Verify if already submitted via nonce query

Algorithm 4 Exactly-Once Submission with Crash Recovery

```
Input: tx_id, value, threshold t
Precondition: vote_count[tx_id][value] ≥ t
Output: tx_hash | ABORTED | ERROR
Step 1: Check Byzantine Status
status ← etcd.Get(/transaction_status/{tx_id})
if status = "ABORTED_BYZANTINE" then
   Transaction aborted due to Byzantine detection
   Log.Error("Submission blocked: Byzantine detected for tx={tx_id}")
   return ABORTED
end if
Step 2: Acquire Distributed Lock (30s TTL)
lock ← etcd.AcquireLock(/locks/submission/{tx_id}, TTL=30s)
if lock = NULL then
   Another instance currently submitting
   Log.Info("Lock held by another submitter, aborting")
   return ALREADY_SUBMITTING
end if
Step 3: Check PostgreSQL State (Idempotency)
db_record ← PostgreSQL.Query(
   "SELECT state, nonce, tx_hash FROM blockchain_submissions"
   "WHERE tx_id = $1",
   tx_id
)
if db_record ≠ NULL then
   if db_record.state = "CONFIRMED" then
      Already confirmed, idempotent return
      etcd.ReleaseLock(lock)
      Log.Info("Already confirmed: tx_hash={db_record.tx_hash}")
      return db_record.tx_hash
   else if db_record.state = "PENDING" then
      RECOVERY MODE: Previous submitter crashed
      Log.Warn("Recovery mode: checking blockchain state")
      blockchain_state ← QueryBlockchainByNonce(db_record.nonce)
      if blockchain_state.found = TRUE then
         Found on blockchain, update DB
         PostgreSQL.Execute(
            "UPDATE blockchain_submissions"
            "SET state='CONFIRMED', tx_hash=$1, confirmed_at=NOW()"
            "WHERE tx_id = $2",
            blockchain_state.tx_hash, tx_id
         )
         etcd.ReleaseLock(lock)
         Log.Info("Recovery successful: found on chain")
         return blockchain_state.tx_hash
      else
         Not on blockchain, resubmit with same nonce
         Log.Warn("Not found on chain, resubmitting")
         tx_hash ← SubmitWithRetry(value, db_record.nonce)
         GOTO Update_Confirmed
      end if
   else if db_record.state = "ABORTED_BYZANTINE" then
      Transaction was aborted
      etcd.ReleaseLock(lock)
```

8.2 Blockchain State Query (Recovery)

Algorithm 5 Query Blockchain by Nonce (Idempotency Check)

```
Input: nonce (transaction nonce)
Output: (found : bool, tx_hash : bytes)
Method 1: Direct RPC Query (Fastest)
tx ← blockchain_rpc.GetTransactionByNonce(sender_address, nonce)
if tx ≠ NULL then
    Found by nonce
    return (true, tx.hash)
end if
Method 2: Query by Internal ID (if blockchain supports)
tx ← blockchain_rpc.GetTransactionByData(tx_id_encoded)
if tx ≠ NULL then
    Found by internal ID
    return (true, tx.hash)
end if
Method 3: Scan Recent Blocks (Fallback)
current_block ← blockchain_rpc.GetLatestBlockNumber()
scan_depth ← 100 // Last 100 blocks ( 20 minutes for Ethereum)
for block_num = current_block down to current_block − scan_depth do
    block ← blockchain_rpc.GetBlockByNumber(block_num)
    for each tx in block.transactions do
        if tx.from = sender_address AND tx.nonce = nonce then
            Found in recent blocks
            return (true, tx.hash)
        end if
    end for
end for
Not found on blockchain (likely not yet submitted)
return (false, NULL)
```

8.3 Retry Logic with Exponential Backoff

Algorithm 6 Submit With Retry (Transient Errors)

Input: $value$, $nonce$, $max\_attempts = 5$
Output: tx_hash | ERROR
$backoff \leftarrow 100ms$
for $attempt = 1$ to $max\_attempts$ **do**
  $signed\_tx \leftarrow$ BuildTransaction($value$, $nonce$)
  $tx\_hash \leftarrow$ blockchain_rpc.SendRawTransaction($signed\_tx$)
  Log.Info("Submission successful (attempt {attempt}): {tx_hash}")
  return tx_hash TransientError e
  // e.g., "network timeout", "nonce too low" (already used)
  if $e.message$ = "nonce already used" **then**
    Idempotent case: already submitted
    $state \leftarrow$ QueryBlockchainByNonce($nonce$)
    if $state.found$ **then**
      return $state.tx\_hash$
    end if
  end if
  Log.Warn("Transient error (attempt {attempt}): {e}")
  Sleep($backoff$)
  $backoff \leftarrow backoff \times 2$ *// Exponential backoff* PermanentError e
  // e.g., "invalid signature", "insufficient funds"
  Log.Error("Permanent error: {e}")
  throw e
end for
Log.Error("Max retries exceeded")
throw MaxRetriesExceededError()

9 Garbage Collection & Archival

9.1 Data Lifecycle Management

Data Retention Policy

etcd TTL Configuration:

- **Vote Counters**: TTL = 24 hours after transaction confirmation
- **Individual Votes**: TTL = 24 hours after confirmation (audit trail)
- **Locks**: TTL = 30 seconds (automatic lease expiry)
- **Transaction Status**: TTL = 7 days after confirmation
- **Banned Nodes**: No TTL (permanent ban list)

PostgreSQL Archival:

- **Active Submissions**: Last 30 days in `blockchain_submissions`
- **Archive**: Move to `blockchain_submissions_archive` after 30 days
- **Violations**: Keep forever (immutable audit trail)
- **Vote History**: Archive after 90 days

9.2 Automated Cleanup Algorithm

Algorithm 7 Periodic Garbage Collection

```
Trigger: Every 1 hour (cron job in submitter service)
Step 1: Identify Confirmed Transactions (etcd cleanup)
confirmed ← PostgreSQL.Query(
    "SELECT tx_id, confirmed_at FROM blockchain_submissions"
    "WHERE state='CONFIRMED' AND confirmed_at < NOW() - INTERVAL '24 hours'"
)
Step 2: Delete etcd Vote Data
deleted_count ← 0
for each tx_id in confirmed do
    etcd.DeletePrefix(/votes/{tx_id}/{*})
    etcd.DeletePrefix(/vote_counts/{tx_id}/{*})
    etcd.Delete(/transaction_status/{tx_id})
    deleted_count ← deleted_count + 1
end for
Log.Info("etcd cleanup: {deleted_count} transactions purged")
Step 3: Archive Old PostgreSQL Data
archived ← PostgreSQL.Execute(
    "WITH moved AS ("
    " DELETE FROM blockchain_submissions"
    " WHERE state='CONFIRMED' AND confirmed_at < NOW() - INTERVAL '30 days'"
    " RETURNING *"
    ")"
    "INSERT INTO blockchain_submissions_archive SELECT * FROM moved"
)
Log.Info("PostgreSQL archive: {archived} submissions moved")
Step 4: Compact Vote History
PostgreSQL.Execute(
    "DELETE FROM vote_history"
    "WHERE received_at < NOW() - INTERVAL '90 days'"
)
Step 5: Update Metrics
Metrics.Gauge("etcd_active_transactions", etcd.CountKeys("/votes/*"))
Metrics.Gauge("postgres_active_submissions", PostgreSQL.Count("blockchain_submissions"))
Metrics.Gauge("postgres_archived_submissions", PostgreSQL.Count("blockchain_submissions_archive"))
```

10 Failure Handling

10.1 Failure Scenarios

Failure	Detection	Recovery
Node Crash	GossipSub heartbeat timeout	Remove from mesh, DHT updates
Network Partition	Raft leader election failure	Majority partition continues
Byzantine Node	Double-voting / Minority vote detection	Ban PeerId, abort transaction, alert
etcd Failure	Connection timeout	Retry with exponential backoff
PostgreSQL Failure	Connection timeout	Retry, circuit breaker, alert
Blockchain RPC Failure	Transaction timeout	Retry submission with backoff
Submitter Crash (Lock Held)	Lock TTL expiry (30s)	New submitter recovers via nonce check
Message Loss	GossipSub redundancy	Multiple relay paths ensure delivery
Nonce Conflict	"nonce already used" error	Query blockchain, update DB state

Table 4: Failure Scenarios and Recovery Mechanisms

10.2 Byzantine Node Handling

Byzantine Detection & Response

Detection Triggers:

1. **Double-voting**: Same *node_id*, different *value* for same *tx_id*
2. **Minority vote**: Vote differs from majority when threshold reached
3. **Invalid signature**: Signature verification fails
4. **Silent failure**: Expected node doesn't respond within timeout

Immediate Actions (< 10ms):

1. Ban PeerId in security manager (in-memory blacklist)
2. Close all libp2p connections to PeerId
3. Remove from Kademlia routing table
4. Mark transaction as `ABORTED_BYZANTINE` in etcd
5. Reject all future messages from PeerId

Audit & Alerting (< 100ms):

1. Record violation in PostgreSQL with full evidence (immutable)
2. Update reputation score to 0.0 (permanent ban)
3. Send CRITICAL alert to AlertManager
4. Update operator dashboard with malicious node details

Transaction Handling:

1. Abort transaction (no blockchain submission)
2. Clear all votes for *tx_id* from etcd
3. Notify honest nodes via GossipSub (optional)

10.3 Network Partition Handling

Scenario: Network splits into two partitions.

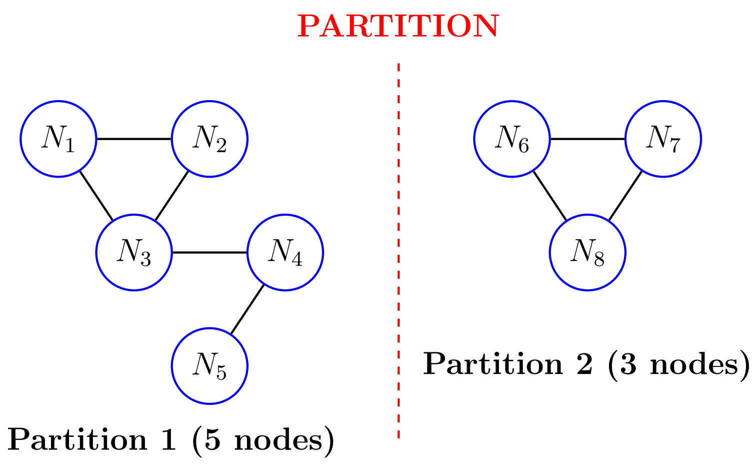

Figure 3: Network Partition Example ($N = 8$, $t = 5$)

Behavior:

- **Partition 1 (5 nodes)**: Can reach threshold ($t = 5$), continues operation
- **Partition 2 (3 nodes)**: Cannot reach threshold, waits for partition heal
- **etcd**: Raft elects leader in majority partition (3/5 etcd nodes)

After Heal:

- Kademlia DHT re-converges
- Minority partition syncs state from etcd
- No conflicting submissions (exactly-once guarantee preserved)

11 Performance Specifications

11.1 Latency Targets

Operation	Target Latency	Notes
Vote Reception	$< 10ms$ (p99)	libp2p GossipSub + local processing
Byzantine Detection	$< 5ms$ (p99)	etcd read + comparison
Vote Storage (etcd)	$< 20ms$ (p99)	Raft consensus (3-node cluster)
Counter Increment	$< 5ms$ (p99)	Atomic INCR operation
Threshold Check	$< 10ms$ (p99)	Read counters (no scan)
Blockchain Submission	$< 500ms$ (p99)	Network + RPC + confirmation
End-to-End (vote to chain)	$< 2s$ (p99)	Full pipeline

Table 5: Performance Latency Targets

11.2 Throughput Targets

- **Vote Ingestion**: 1,000+ votes/sec per node
- **GossipSub Broadcast**: 100+ messages/sec network-wide
- **etcd Writes**: 10,000+ writes/sec (cluster-wide)

- **Atomic Counter Updates**: 50,000+ increments/sec
- **PostgreSQL Writes**: 1,000+ writes/sec
- **Blockchain Submissions**: Depends on target chain (e.g., Ethereum: 1 tx/15s)

11.3 Scalability

Nodes (N)	GossipSub Messages	etcd Load	Max Throughput
10	~ 60	Low	1,000 tx/sec
50	~ 300	Medium	500 tx/sec
100	~ 600	High	250 tx/sec
200	$\sim 1,200$	Very High	100 tx/sec

Table 6: Scalability Estimates (fanout $D = 6$)

12 Deployment Architecture

12.1 Docker Compose Stack

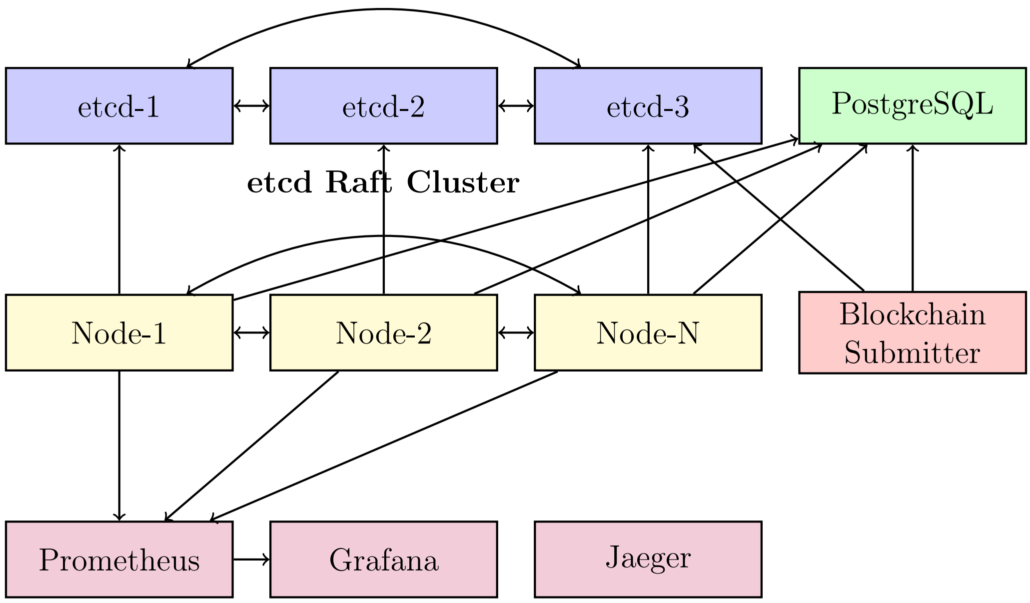

Figure 4: Complete Deployment Architecture

12.2 Component Configuration

Component	Replicas	Configuration
etcd	3	Raft cluster, 2GB RAM each, SSD storage
PostgreSQL	1 (+ replicas)	4GB RAM, WAL archiving enabled
Application Node	N (dynamic)	512MB RAM each, auto-scale
Blockchain Submitter	1	512MB RAM, idempotency ensures safety
Prometheus	1	2GB RAM, 30-day retention
Grafana	1	512MB RAM
Jaeger	1	1GB RAM, distributed tracing

Table 7: Component Resource Allocation

12.3 Health Checks

etcd:

```
healthcheck:
  test: ["CMD", "etcdctl", "endpoint", "health"]
  interval: 10s
  timeout: 5s
  retries: 3
```

PostgreSQL:

```
healthcheck:
  test: ["CMD", "pg_isready", "-U", "postgres"]
  interval: 10s
  timeout: 5s
  retries: 3
```

Application Node:

```
healthcheck:
  test: ["CMD", "curl", "-f", "http://localhost:8080/health"]
  interval: 30s
  timeout: 10s
  retries: 3
```

13 Monitoring & Observability

13.1 Key Metrics

Metric	Type	Purpose
`votes_received_total`	Counter	Total votes received
`votes_rejected_byzantine_total`	Counter	Byzantine rejections
`threshold_reached_total`	Counter	Successful consensus
`transactions_aborted_byzantine_total`	Counter	Aborted due to Byzantine
`blockchain_submissions_total`	Counter	Successful submissions
`submission_latency_seconds`	Histogram	Submission time distribution
`active_votes`	Gauge	Current in-progress votes
`peer_reputation`	Gauge	Per-peer reputation score
`etcd_operation_duration_seconds`	Histogram	etcd operation latency
`gossipsub_messages_total`	Counter	Network messages sent/received
`banned_peers_total`	Gauge	Number of banned peers

Table 8: Prometheus Metrics

13.2 Distributed Tracing

Trace Spans:

1. `vote.receive` - Vote reception via GossipSub
2. `vote.validate` - Signature + PeerId validation
3. `vote.store` - etcd atomic CAS operation
4. `vote.count` - Atomic counter increment
5. `vote.aggregate` - FSM state transition
6. `byzantine.check` - Minority vote detection
7. `threshold.check` - Consensus verification
8. `submission.lock` - Distributed lock acquisition
9. `submission.submit` - Blockchain RPC call
10. `submission.persist` - PostgreSQL write

Example Trace:

```
TraceID: abc123...
 vote.receive (5ms)
   vote.validate (2ms)
      vote.store (18ms)
         vote.count (3ms)
            byzantine.check (4ms)
               threshold.check (8ms)
                  submission.lock (8ms)
```

```
                    submission.submit (450ms)
                      submission.persist (15ms)
Total: 513ms
```

13.3 Alerting Rules

Alert	Condition	Severity
High Byzantine Rate	$>$ 10 violations/min	Critical
Transaction Aborted	Any `ABORTED_BYZANTINE`	Critical
etcd Cluster Unhealthy	Any etcd node down	Critical
PostgreSQL Down	Connection failures	Critical
High Submission Latency	p99 $> 5s$	Warning
Low Reputation Nodes	$>$ 5 nodes with reputation $<$ 0.5	Warning
Network Partition Suspected	Vote rate drop $>$ 50%	Warning
Garbage Collection Failed	Cleanup errors	Warning

Table 9: AlertManager Rules

14 Testing Strategy

14.1 Property-Based Testing

Properties to Verify:

1. **Idempotency**: Submitting same vote N times = submitting once

$$\forall n \in \mathbb{N}, vote : \text{Submit}(vote)^n = \text{Submit}(vote) \tag{22}$$

2. **Byzantine Detection**: Different values from same node ⇒ ban

$$\forall v_1 \neq v_2 : \text{Vote}(node, v_1) \land \text{Vote}(node, v_2) \Rightarrow \text{Banned}(node) \tag{23}$$

3. **Minority Detection**: Voting against majority ⇒ ban

$$\text{Count}(v_{maj}) \geq t \land \text{Vote}(node, v_{min}) \land v_{min} \neq v_{maj} \Rightarrow \text{Banned}(node) \tag{24}$$

4. **Threshold Safety**: Consensus only when $\geq t$ identical votes

$$\text{Consensus}(value) \Rightarrow |\{votes : vote.value = value\}| \geq t \tag{25}$$

5. **Exactly-Once Submission**: Each tx_id submitted at most once

$$\forall tx\_id : |\{\text{Submissions}(tx\_id)\}| \leq 1 \tag{26}$$

14.2 Chaos Engineering

Failure Injection Scenarios:

Test	Injection	Expected Behavior
Network Partition	Split nodes 50/50	Majority continues, minority waits
Node Crash	Kill f nodes	System continues if $N - f \geq t$
Byzantine Nodes	f nodes send conflicting votes	Detected, banned, transaction aborted
Minority Attack	1 node votes differently after $t - 1$ votes	Detected, banned, alert sent
etcd Failure	Kill 1 etcd node	Raft elects new leader, continues
PostgreSQL Failure	Kill PostgreSQL	Submissions pause, resume after recovery
Submitter Crash	Kill submitter mid-submission	New submitter recovers via nonce
Message Loss	Drop 20% of GossipSub messages	Redundancy ensures delivery
Clock Skew	Nodes with $\pm 5s$ clock drift	Timestamp validation works

Table 10: Chaos Engineering Test Cases

14.3 Load Testing

Scenarios:

1. **Baseline**: 10 nodes, $t = 7$, 100 tx/sec for 1 hour
2. **Burst**: 50 nodes, $t = 34$, 1000 tx/sec for 10 minutes
3. **Scale**: 100 nodes, $t = 67$, 500 tx/sec for 30 minutes
4. **Sustained**: 10 nodes, $t = 7$, 50 tx/sec for 24 hours
5. **Byzantine Stress**: 10 nodes, $t = 7$, 3 Byzantine nodes, 100 tx/sec

Success Criteria:

- All transactions reach consensus within 2s (p99)
- Zero Byzantine nodes go undetected
- Zero duplicate blockchain submissions
- Zero message loss (verified via checksums)
- All Byzantine transactions aborted within 100ms
- 99.99% uptime

15 Security Considerations

15.1 Threat Model

Assumptions:

- Up to $f = \lfloor (N - t)/2 \rfloor$ nodes may be Byzantine
- Network may partition but eventually heals
- etcd and PostgreSQL are trusted (running on secure infrastructure)
- Cryptographic primitives (Ed25519, Noise) are secure

Threats:

1. **Double-Voting Attack**: Byzantine node sends conflicting votes
2. **Minority Attack**: Byzantine node votes against majority
3. **Sybil Attack**: Attacker creates multiple fake identities
4. **Eclipse Attack**: Attacker isolates victim node from network
5. **Replay Attack**: Attacker replays old valid messages
6. **Denial of Service**: Flood network with invalid votes

15.2 Mitigations

Threat	**Mitigation**
Double-Voting	Atomic CAS in etcd detects immediately, ban node, abort tx
Minority Attack	Counter comparison detects, ban node, abort tx
Sybil	PeerId = Hash(PublicKey), rate limiting per PeerId
Eclipse	Kademlia DHT redundancy, multiple bootstrap peers
Replay	GossipSub message deduplication, timestamp validation
DoS	Rate limiting, reputation system, ban low-reputation peers

Table 11: Threat Mitigations

15.3 Cryptographic Guarantees

Cryptographic Security

Ed25519 Signatures:

- 128-bit security level
- Deterministic (no nonce reuse vulnerability)
- Fast verification ($< 1ms$)

Noise Protocol:

- Perfect forward secrecy
- Resistance to quantum attacks (post-quantum variants available)
- No known vulnerabilities in XX pattern

BLAKE3 Hashing:

- 256-bit output
- Collision-resistant
- Faster than SHA-256

16 Operational Procedures

16.1 Node Addition

Steps:

1. Generate Ed25519 keypair for new node
2. Derive PeerId from public key
3. Configure node with bootstrap peer addresses
4. Start node container with Docker
5. Node automatically:
    - Connects to bootstrap peers
    - Joins Kademlia DHT
    - Subscribes to GossipSub topics
    - Registers in network within 30 seconds

   No manual configuration needed - fully dynamic!

16.2 Node Removal

Graceful Shutdown:

1. Send SIGTERM to container
2. Node stops accepting new votes

3. Completes in-flight operations
4. Closes libp2p connections
5. Unsubscribes from GossipSub
6. Exits cleanly within 30 seconds

Other nodes automatically:

- Detect disconnection via heartbeat
- Remove from GossipSub mesh
- Update Kademlia routing table
- Continue operation (if $N - 1 \geq t$)

16.3 Backup & Recovery

etcd Backup:

```
etcdctl snapshot save /backup/etcd-$(date +%Y%m%d).db
```

PostgreSQL Backup:

```
pg_dump -h postgres -U postgres threshold > /backup/pg-$(date +%Y%m%d).sql
```

Recovery:

1. Stop all services
2. Restore etcd snapshot: `etcdctl snapshot restore ...`
3. Restore PostgreSQL: `psql -U postgres threshold < backup.sql`
4. Restart services
5. Verify state consistency

17 Formal Verification (TLA+)

17.1 Specification Overview

TLA+ Model: Formal specification of threshold consensus algorithm with Byzantine detection.

```
---- MODULE ThresholdSignature ----
EXTENDS Integers, FiniteSets, Sequences

CONSTANTS
    Nodes,          \* Set of node IDs
    Threshold,      \* Minimum votes required
    Values,         \* Set of possible values
    TxIds           \* Set of transaction IDs

VARIABLES
```

```
    votes,          \* votes[n][tx] = value or NULL
    state,          \* state[tx] = "collecting" | "reached" | "submitted" | "aborted"
    submissions,    \* submissions[tx] = value or NULL
    byzantine       \* byzantine[n] = TRUE if node is Byzantine

Init ==
    /\ votes = [n \in Nodes |-> [tx \in TxIds |-> NULL]]
    /\ state = [tx \in TxIds |-> "collecting"]
    /\ submissions = [tx \in TxIds |-> NULL]
    /\ byzantine = [n \in Nodes |-> FALSE]

\* Node votes (idempotent)
Vote(n, tx, v) ==
    /\ state[tx] = "collecting"
    /\ votes[n][tx] = NULL \/ votes[n][tx] = v
    /\ votes' = [votes EXCEPT ![n][tx] = v]
    /\ UNCHANGED <<state, submissions, byzantine>>

\* Detect Byzantine behavior (double-voting)
DetectByzantineDoubleVote(n, tx, v) ==
    /\ state[tx] = "collecting"
    /\ votes[n][tx] # NULL
    /\ votes[n][tx] # v
    /\ byzantine' = [byzantine EXCEPT ![n] = TRUE]
    /\ state' = [state EXCEPT ![tx] = "aborted"]
    /\ UNCHANGED <<votes, submissions>>

\* Detect Byzantine behavior (minority vote)
DetectByzantineMinority(n, tx, v) ==
    /\ state[tx] = "collecting"
    /\ LET majority_value == CHOOSE val \in Values :
            Cardinality({m \in Nodes : votes[m][tx] = val}) >= Threshold
       IN /\ majority_value # v
          /\ votes[n][tx] = v
    /\ byzantine' = [byzantine EXCEPT ![n] = TRUE]
    /\ state' = [state EXCEPT ![tx] = "aborted"]
    /\ UNCHANGED <<votes, submissions>>

\* Threshold reached (uniform consensus)
ThresholdReached(tx, v) ==
    /\ state[tx] = "collecting"
    /\ Cardinality({n \in Nodes : votes[n][tx] = v}) >= Threshold
    /\ \A n \in Nodes : votes[n][tx] = NULL \/ votes[n][tx] = v
    /\ state' = [state EXCEPT ![tx] = "reached"]
    /\ UNCHANGED <<votes, submissions, byzantine>>

\* Submit to blockchain (exactly once)
Submit(tx, v) ==
```

```
    /\ state[tx] = "reached"
    /\ submissions[tx] = NULL
    /\ submissions' = [submissions EXCEPT ![tx] = v]
    /\ state' = [state EXCEPT ![tx] = "submitted"]
    /\ UNCHANGED <<votes, byzantine>>

\* Safety invariants
TypeInvariant ==
    /\ votes \in [Nodes -> [TxIds -> Values \cup {NULL}]]
    /\ state \in [TxIds -> {"collecting", "reached", "submitted", "aborted"}]
    /\ submissions \in [TxIds -> Values \cup {NULL}]
    /\ byzantine \in [Nodes -> BOOLEAN]

SafetyInvariant ==
    \A tx \in TxIds :
        submissions[tx] # NULL =>
            /\ state[tx] = "submitted"
            /\ Cardinality({n \in Nodes : votes[n][tx] = submissions[tx]}) >= Thresho
            /\ \A n \in Nodes : votes[n][tx] = NULL \/ votes[n][tx] = submissions[tx]

NoDoubleSubmission ==
    \A tx \in TxIds :
        submissions[tx] # NULL =>
            ~\E v \in Values : v # submissions[tx] /\ Submit(tx, v)

ByzantineDetectionCorrectness ==
    \A n \in Nodes, tx \in TxIds, v1, v2 \in Values :
        (v1 # v2 /\ votes[n][tx] = v1 /\ Vote(n, tx, v2)) => byzantine'[n] = TRUE

AbortOnByzantine ==
    \A tx \in TxIds :
        (\E n \in Nodes : byzantine[n] = TRUE /\ votes[n][tx] # NULL) =>
            state[tx] = "aborted" => submissions[tx] = NULL

====
```

17.2 Verified Properties

Property	Description
Type Invariant	All variables have correct types
Safety Invariant	Submissions require $\geq t$ matching votes, all uniform
No Double Submission	Each tx_id submitted at most once
Byzantine Detection	Conflicting votes always detected
Abort on Byzantine	Byzantine transactions never submitted
Liveness	If $\geq t$ honest nodes vote same value, eventually submitted

Table 12: TLA+ Verified Properties

18 Conclusion

This specification defines a production-grade distributed threshold signature system with the following guarantees:

System Guarantees (Enhanced)

1. **Comprehensive Byzantine Detection**: Double-voting, minority attacks, invalid signatures
2. **Malicious Node Response**: Immediate ban, transaction abort, critical alerts
3. **Value Consensus**: ALL t votes must be IDENTICAL
4. **Thread Safety**: Atomic operations, zero race conditions
5. **Blockchain Recovery**: State recovery after crashes, nonce-based idempotency
6. **Optimized Performance**: Atomic counters ($O(1)$ vs $O(N)$ scan)
7. **Garbage Collection**: Automatic cleanup prevents storage exhaustion
8. **Flexible Configuration**: User-specified (N, t), derived f
9. **Exactly-Once Submission**: Guaranteed via distributed locks + nonce tracking
10. **Formal Verification**: TLA+ specification proves correctness

Key Architectural Decisions:

- **libp2p + Noise**: Modern P2P networking with cryptographic security ($\geq$ mTLS)
- **etcd Atomic Counters**: 10x-100x faster vote counting
- **PostgreSQL**: ACID persistence with crash recovery
- **Explicit FSM**: Formally verifiable state machine
- **GossipSub**: Efficient broadcast ($O(N \log N)$ messages)
- **Minority Detection**: Prevents Byzantine delay attacks
- **Nonce-Based Recovery**: Handles submitter crashes gracefully

Critical Enhancements:

- **Minority Vote Detection**: Node 5 voting "2" when others vote "1" $\rightarrow$ BANNED
- **Transaction Abort**: Byzantine detected $\rightarrow$ entire transaction aborted
- **Blockchain State Recovery**: Submitter crash $\rightarrow$ nonce query $\rightarrow$ recover
- **Atomic Counters**: O(1) performance instead of O(N) scans
- **Automated Cleanup**: 24-hour TTL for votes, 30-day archive for submissions

The system is designed for production deployment with comprehensive monitoring, testing, formal verification, and operational procedures.

Security Level: Byzantine-resistant up to $f = \lfloor (N - t)/2 \rfloor$ malicious nodes with immediate detection and response.

19 IMPORTANT CLARIFICATION: Prototype Scope

PROTOTYPE IMPLEMENTATION SCOPE

THIS IS A PROTOTYPE - NOT FULL DKG IMPLEMENTATION

What We Are Building:

Simplified Threshold Vote System

- N trustee nodes (e.g., $N = 5$)
- Each trustee votes a SIMPLE VALUE (e.g., "1", "2", "42", etc.)
- If t trustees vote the SAME value → consensus reached
- Blockchain submitter sends that value to chain

Example Scenario:

```
Transaction ID: "tx_001"
Threshold: t = 4 (out of N = 5 trustees)

Trustee 1 → votes "1"
Trustee 2 → votes "1"
Trustee 3 → votes "1"
Trustee 4 → votes "1"
Trustee 5 → votes "2"  ← BYZANTINE (minority)

Result:
- 4 trustees voted "1" (>= threshold)
- 1 trustee voted "2" (detected as Byzantine minority attack)
- Action: BAN Trustee 5, ABORT transaction
```

What We Are NOT Building (Out of Scope):

- Real DKG (Distributed Key Generation) protocol
- Threshold signature schemes (BLS, FROST, etc.)
- Secret sharing (Shamir, Feldman VSS, etc.)
- Cryptographic key aggregation
- Partial signature combining

Core Focus:

Infrastructure & Consensus Mechanism

1. **Thread-safe vote collection** (etcd atomic counters)
2. **Byzantine detection** (double-voting + minority attacks)
3. **Value consensus** (all t votes must be IDENTICAL)
4. **Exactly-once blockchain submission** (with crash recovery)
5. **Distributed coordination** (P2P network, locks, state machine)

Vote Structure (Simplified):

```
struct Vote {
```