Bitcoin Lending Protocol

Technical Specification Document

Phase 2: Bitcoin-Native Custody with Threshold Signatures

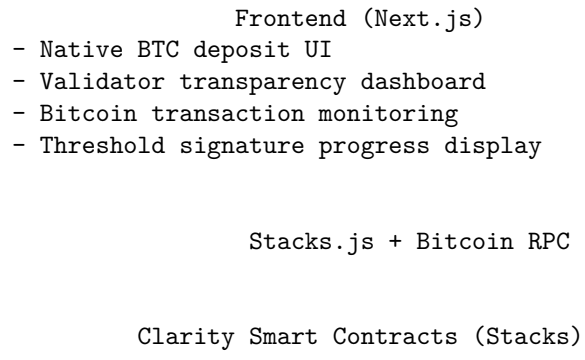
Version: 1.2
Date: 12th January 2026
Status: Draft for Review
Author: Jamie (Project Lead)
Builds On: Phase 1 (Stacks/sBTC Implementation)

Table of Contents

---

1. Architecture Overview

1.1 System Components



```
loan-protocol-v2.clar
- Bitcoin deposit monitoring
- Validator action requests
- Cross-chain state sync
- Auction logic (unchanged)
- Multi-stablecoin (unchanged)
```

```
                Validator Consensus Protocol

            Validator Network (15 nodes)

Validator     Validator     Validator
    1             2             3        ... (15)

 - BTC         - BTC         - BTC
 - Stacks      - Stacks      - Stacks
 - FROST       - FROST       - FROST
 - HSM         - HSM         - HSM

            libp2p gossip

                 Threshold Signatures (10-of-15)

                Bitcoin Blockchain

  Threshold Multisig Addresses (Taproot)
  - One address per loan
  - Public key from DKG
  - Controlled by validator network
  - No single point of failure
```

1.2 Phase 2 Loan Flow (Native Bitcoin)

Step 1: Loan Creation with Bitcoin Address Generation

```
User → Frontend
  1. User creates loan request (amount, duration, stablecoin)
  2. Frontend calls Stacks contract

Frontend → Stacks Contract
  3. create-loan-auction-v2(loan-params)
  4. Contract generates unique loan-id
  5. Contract emits "bitcoin-address-needed" event

Validators Monitor Stacks
  6. All validators see "bitcoin-address-needed" event
  7. Validators derive deterministic Bitcoin address:
     - Use loan-id as derivation path
     - Combined public key P = P_1 + P_2 + ... + P_15
     - Taproot address from combined key
  8. Validators submit Bitcoin address to Stacks contract

Stacks Contract
  9. Contract receives Bitcoin address from validators
  10. Contract stores: loan → bitcoin_address mapping
  11. Contract emits "deposit-ready" event with address

Frontend
  12. Display Bitcoin address (QR + text) to user
  13. User sends BTC from their wallet
  14. Frontend monitors Bitcoin mempool
```

Step 2: Bitcoin Deposit Confirmation

```
Bitcoin Network
  1. User's BTC transaction broadcasts
  2. Transaction appears in mempool
  3. Miners include in block
  4. Transaction gets confirmations (target: 3+)

Validators Monitor Bitcoin
  5. All validators watch threshold addresses
  6. Validators see deposit transaction
  7. Wait for 3 confirmations

Validators → Stacks
  8. Validators submit Bitcoin proof to Stacks:
     - Transaction ID
```

```
     - Block height
     - Merkle proof
     - Output amount
  9. Multiple validators submit (redundancy)

Stacks Contract
  10. Contract receives Bitcoin proofs
  11. Contract verifies:
      - Transaction is real (merkle proof)
      - Amount matches loan requirement
      - Confirmations >= 3
  12. Contract activates loan
  13. Auction begins on Stacks
```

Step 3: Auction & Loan Finalization (Same as Phase 1)

```
Stacks Layer
  - Lenders bid in stablecoin (competitive bidding)
  - Lowest bid wins
  - Winner transfers stablecoin on Stacks
  - Loan status: "active"

No validator action needed yet - BTC stays locked
```

Step 4: Bitcoin Release to Borrower

```
Stacks Contract
  1. Auction finalizes, winner determined
  2. Winner transfers stablecoin to borrower (on Stacks)
  3. Contract emits "release-collateral" action request:
     {
       loan-id: 42,
       action: "release",
       bitcoin-txid: "abc123...",
       recipient: "bc1q...", // Borrower's Bitcoin address
       amount: 150000000, // 1.5 BTC in satoshis
       fee: 50000 // Estimated fee
     }

Validators See Action Request
  4. All validators monitor Stacks for action requests
  5. Each validator independently verifies:
     - Stacks contract state is correct
     - Auction properly finalized
     - Winner paid stablecoin
     - Recipient address is valid
  6. If valid, validator creates partial signature
```

```
Threshold Signing
  7. Validator creates partial signature using key share
  8. Validator broadcasts partial signature to network
  9. Coordinator collects partial signatures
  10. Once 10+ received, coordinator aggregates
  11. Final Schnorr signature is valid for threshold address

Bitcoin Transaction
  12. Coordinator constructs Bitcoin transaction:
      Input: Threshold address UTXO
      Output: Borrower address (1.5 BTC - fee)
  13. Coordinator attaches aggregated signature
  14. Coordinator broadcasts to Bitcoin network
  15. Transaction confirms in Bitcoin block

Stacks Contract Update
  16. Validators submit Bitcoin confirmation proof
  17. Contract updates loan status
  18. Frontend shows BTC received by borrower
```

Step 5: Repayment & Collateral Return

```
Borrower Repays on Stacks
  1. Borrower transfers stablecoin to lender (Stacks)
  2. Contract verifies repayment
  3. Contract emits "release-collateral" action request
     (same process as Step 4)

Validators Release Collateral
  4. Threshold signing process (same as Step 4)
  5. BTC transfers from threshold address to borrower
  6. Transaction confirms on Bitcoin
  7. Loan complete, NFTs burned
```

Step 6: Default Scenario

```
If Borrower Doesn't Repay
  1. Loan maturity block reached on Stacks
  2. Lender calls claim-collateral on Stacks
  3. Contract emits "claim-collateral" action request
     (BTC to lender's address instead of borrower)

Validators Process Claim
  4. Threshold signing with lender as recipient
  5. BTC transfers to lender
  6. Loan status: "defaulted"
```

1.3 Key Differences from Phase 1

Aspect	Phase 1 (sBTC)	Phase 2 (Native BTC)
Collateral Asset	sBTC on Stacks	Actual Bitcoin on Bitcoin
Collateral Location	Stacks contract	Threshold multisig on Bitcoin
Deposit Process	Single Stacks transaction	Send BTC to address, wait confirmations
Custody	Stacks contract	Distributed validator network
Release Mechanism	Stacks transaction	Threshold signatures + Bitcoin transaction
Trust Model	sBTC peg operators	10-of-15 validator threshold
Confirmation Time	~10 min (Stacks)	~30 min (3 Bitcoin confirmations)
Fees	Stacks gas (~$1-5)	Bitcoin fees (~$5-50 depending on network)

---

2. Threshold Cryptography Specifications

2.1 Cryptographic Scheme Selection

Chosen Scheme: FROST (Flexible Round-Optimized Schnorr Threshold Signatures)

Why FROST? - Schnorr-based (Bitcoin Taproot compatible) - Non-interactive after DKG (2-round signing only) - Efficient aggregation - Provably secure - Production-ready implementations available

Alternative Considered: MuSig2 - Also Schnorr-based - Requires all participants online simultaneously - Less flexible than FROST for 10-of-15 threshold

Threshold Configuration: **10-of-15** - 15 total validators - 10 signatures required (67% threshold) - 5 validators can be offline - 6+ colluding validators needed to steal (unlikely)

2.2 Distributed Key Generation (DKG)

Purpose: Generate shared private key without any single party knowing it.

Process:

```
Round 0: Setup

- Coordinator announces DKG ceremony
```

```
- All 15 validators join
- Parameters: n=15, t=10 (threshold)
- Each validator generates random secret s_i

Round 1: Commitment Phase (VSS)

Each validator i:
  1. Creates secret polynomial: f_i(x) = s_i + a_1*x + ... + a_9*x^9
     (degree 9 polynomial for t=10)
  2. Computes commitments: C_ij = g^(a_j) for j=0..9
  3. Broadcasts commitments to all validators

Round 2: Share Distribution

Each validator i:
  1. Evaluates polynomial at each validator's ID:
     share_ij = f_i(j) for j=1..15
  2. Encrypts share_ij with validator j's public key
  3. Sends encrypted shares to each validator

Round 3: Share Verification

Each validator j:
  1. Receives shares from all other validators
  2. Decrypts own share from each validator i
  3. Verifies share against commitments:
     g^(share_ij) = C_i0 * C_i1^j * C_i2^(j^2) * ... * C_i9^(j^9)
  4. If verification fails, broadcast complaint
  5. If valid, accept share

Round 4: Complaint Handling

If complaints exist:
  1. Accused validator reveals shares to everyone
  2. All validators verify revealed shares
  3. If invalid, exclude malicious validator
  4. If valid, exclude complaining validator

Round 5: Key Derivation

Each validator j:
  1. Combines all received shares:
     secret_share_j = sum(share_ij for all i)
  2. Computes public key share:
     P_j = g^(secret_share_j)
```

```
All validators:
  1. Combine public key shares:
     P = P_1 + P_2 + ... + P_15
  2. P is the combined public key
  3. Bitcoin address derived from P (Taproot)

Result:

  Each validator has secret_share_j (kept private)
  Combined public key P is known to everyone
  No validator knows the combined private key
  Any 10 validators can create valid signatures
```

Security Properties: - **Soundness**: Valid shares pass verification - **Privacy**: No information about full private key revealed - **Correctness**: Combined key corresponds to valid threshold - **Robustness**: Handles up to 5 malicious validators

Implementation:

```rust
// Rust pseudocode using FROST library

use frost_secp256k1 as frost;

// Round 1: Generate commitments
let (secret_package, public_package) = frost::keys::dkg::part1(
    validator_id,
    15,  // total participants
    10,  // threshold
    &mut rng
)?;

// Round 2: Generate and send shares
let round2_packages = frost::keys::dkg::part2(
    secret_package,
    &round1_packages_from_all
)?;

// Round 3: Finalize key generation
let (key_package, public_key_package) = frost::keys::dkg::part3(
    &round2_packages_from_all,
    &round1_packages_from_all
)?;

// key_package contains this validator's secret share
// public_key_package contains combined public key
```

2.3 Threshold Signature Generation

Purpose: Create valid Bitcoin signature using 10-of-15 validators.

FROST Signing Process:

```
Phase 1: Nonce Commitment

Coordinator → All Validators:
  - Announces signing request
  - Message: Bitcoin transaction hash to sign
  - Nonce: Unique per signing session

Each Validator (independently):
  1. Generate random nonces: (d_i, e_i)
  2. Compute commitments: (D_i, E_i) = (g^d_i, g^e_i)
  3. Send (D_i, E_i) to coordinator

Coordinator:
  4. Collect commitments from all validators
  5. Wait for at least 10 commitments
  6. Publish list of participants (who will sign)
  7. Compute binding values for each participant

Phase 2: Signature Generation

Coordinator → Participants:
  - List of signing participants (10+)
  - All nonce commitments
  - Message to sign

Each Participating Validator:
  1. Compute Lagrange coefficient  _i
  2. Compute group commitment R
  3. Compute challenge c = H(R || P || m)
  4. Create partial signature:
     z_i = d_i + (e_i *  _i) + ( _i * secret_share_i * c)
  5. Send z_i to coordinator

Coordinator:
  6. Collect partial signatures (need 10+)
  7. Aggregate partial signatures:
     z = sum(z_i for all participants)
  8. Final signature: (R, z)
  9. Verify signature is valid for public key P
  10. If valid, use for Bitcoin transaction
```

```
Phase 3: Bitcoin Transaction

Coordinator:
  1. Construct Bitcoin transaction
  2. Attach signature (R, z) to transaction
  3. Broadcast to Bitcoin network
  4. Monitor for confirmation
```

Timing: - Phase 1: ~30 seconds (validators respond) - Phase 2: ~30 seconds (partial signatures) - Aggregation: <1 second - **Total**: ~1-2 minutes typically

Implementation:

```rust
// Round 1: Generate nonce commitments
let (nonce, commitments) = frost::round1::commit(
    &key_package,
    &mut rng
);

// Round 2: Create partial signature
let signature_share = frost::round2::sign(
    &signing_package,  // From coordinator
    &nonce,
    &key_package
)?;

// Coordinator aggregates
let group_signature = frost::aggregate(
    &signing_package,
    &signature_shares,  // From 10+ validators
    &public_key_package
)?;

// group_signature is a valid Schnorr signature
```

2.4 Address Derivation

Bitcoin Address Generation:

```
Given: Combined public key P from DKG

For Loan ID = N:
  1. Derive child key using BIP32-style derivation:
     P_N = P + H(P || N) * G

  2. Create Taproot output:
     - Internal key: P_N
     - Script tree: None (key-path only)
```

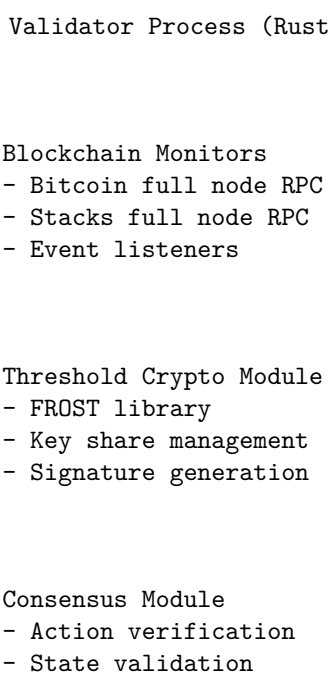
```
  3. Compute Taproot output key:
     Q = P_N + H("TapTweak" || P_N) * G

  4. Create Bech32m address:
     address = bech32m_encode("bc", 1, Q)

Result: bc1p[52 characters]

Example:
  Loan #1 → bc1p8xj2f7a9zmqwn3ktv8s6r4l2c5h9tp3m4d6x8y
  Loan #2 → bc1pq5m9tn2h7r3k8d4f6x9c2v5b8n3m7p4w6s9t2k

Each loan gets unique address, all controlled by same validator set
```

Address Properties: - **Unique per loan**: Deterministic but different - **Taproot**: Single signature appearance (privacy) - **Verifiable**: Anyone can verify address derivation - **Secure**: Requires 10-of-15 to spend

---

3. Validator Node Architecture

3.1 Validator Software Stack



```
- Conflict resolution


Network Module
- libp2p
- Gossip protocol
- Peer discovery


Storage
- HSM/secure enclave
- SQLite (state)
- Logs


Metrics & Monitoring
- Prometheus exporter
- Health checks
- Alerts




   Bitcoin      Stacks      Peer Validators
    Node         Node        (libp2p)
```

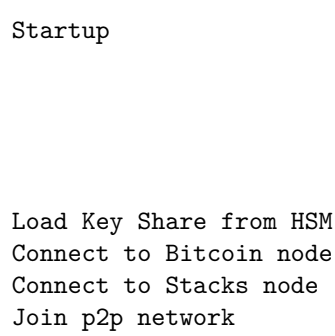
3.2 Validator Configuration

config.toml:

```toml
[validator]
id = 1  # Validator ID (1-15)
network = "mainnet"  # or "testnet"

[bitcoin]
rpc_url = "http://localhost:8332"
rpc_user = "bitcoin"
rpc_password = "secret"
confirmations_required = 3

[stacks]
rpc_url = "http://localhost:20443"
```

```toml
contract_address = "SP1234.loan-protocol-v2"

[threshold]
scheme = "frost"
total_validators = 15
threshold = 10

[key_storage]
type = "hsm"  # or "file" for testing
hsm_slot = 0
hsm_pin_env = "HSM_PIN"

[network]
p2p_listen = "0.0.0.0:9001"
bootstrap_peers = [
  "/ip4/validator1.example.com/tcp/9001/p2p/12D3KooW...",
  "/ip4/validator2.example.com/tcp/9001/p2p/12D3KooW...",
]

[monitoring]
metrics_port = 9090
health_check_port = 8080
log_level = "info"

[coordinator]
enabled = true  # Only one coordinator needed
coordinator_url = "http://coordinator.example.com:8000"
```

3.3 Validator State Machine



Syncing

Bitcoin: Synced?
Stacks: Synced? No

Yes

Ready

(Loop)

Monitor Events
- Bitcoin deposits
- Stacks action requests
- Peer messages

Event Detected?
Yes → Process
No → Wait

Verify Event
- Check Stacks contract
- Validate action
- Ensure no double-sign

Valid? No

```
Yes



Signing
- Generate nonce
- Create partial sig
- Broadcast to coordinator



Wait for Aggregation
(Coordinator combines)



Verify Final Signature
- Check aggregated sig
- Ensure transaction valid



   Back to Ready
```

3.4 Key Storage and Security

HSM Integration (YubiHSM 2):

```rust
use yubihsm::{Client, Connector, Credentials};

// Initialize HSM connection
let connector = Connector::usb(&Default::default())?;
let client = Client::open(connector, Credentials::default(), true)?;

// Store key share (one-time, during DKG)
fn store_key_share(
    client: &Client,
    key_share: &SecretShare
) -> Result<(), Error> {
    // Serialize key share
    let key_data = bincode::serialize(key_share)?;
```

```rust
    // Store in HSM (encrypted at rest)
    client.put_opaque_object(
        VALIDATOR_KEY_SLOT,
        "validator_key_share",
        &key_data,
        &[Capability::ExportableUnderWrap]
    )?;

    Ok(())
}

// Retrieve key share for signing
fn load_key_share(
    client: &Client
) -> Result<SecretShare, Error> {
    // Retrieve from HSM
    let key_data = client.get_opaque_object(VALIDATOR_KEY_SLOT)?;

    // Deserialize
    let key_share = bincode::deserialize(&key_data)?;

    Ok(key_share)
}

// Sign with HSM protection
fn create_partial_signature(
    client: &Client,
    message: &[u8],
    signing_package: &SigningPackage
) -> Result<SignatureShare, Error> {
    // Load key share from HSM
    let key_share = load_key_share(client)?;

    // Generate signature (computation happens in memory)
    let signature = frost::round2::sign(
        signing_package,
        &nonce,
        &key_share
    )?;

    // Zero out key_share immediately
    drop(key_share);

    Ok(signature)
}
```

File-based storage (testnet only):

```rust
use aes_gcm::{Aes256Gcm, Key, Nonce};
use aes_gcm::aead::{Aead, NewAead};

// Store encrypted key share to file
fn store_key_share_file(
    key_share: &SecretShare,
    password: &str
) -> Result<(), Error> {
    // Derive encryption key from password
    let key = derive_key_from_password(password)?;
    let cipher = Aes256Gcm::new(Key::from_slice(&key));

    // Serialize
    let plaintext = bincode::serialize(key_share)?;

    // Encrypt
    let nonce = Nonce::from_slice(b"unique_nonce");
    let ciphertext = cipher.encrypt(nonce, plaintext.as_ref())?;

    // Write to file
    fs::write("key_share.enc", ciphertext)?;

    Ok(())
}

// Load and decrypt key share
fn load_key_share_file(
    password: &str
) -> Result<SecretShare, Error> {
    let key = derive_key_from_password(password)?;
    let cipher = Aes256Gcm::new(Key::from_slice(&key));

    let ciphertext = fs::read("key_share.enc")?;
    let nonce = Nonce::from_slice(b"unique_nonce");
    let plaintext = cipher.decrypt(nonce, ciphertext.as_ref())?;

    let key_share = bincode::deserialize(&plaintext)?;

    Ok(key_share)
}
```

SECTION 3.5: Validator Recruitment & Onboarding (NEW)

Budget: $25,000
Timeline: Months 8-10 (3 months)
Deliverable: D2.1a

3.5.1 Qualification Criteria

Technical Requirements: - Rust programming (1+ years) - Linux sysadmin (2+ years) - 24/7 availability commitment - Network security knowledge - Bitcoin/blockchain experience

Hardware Requirements: - CPU: 8+ cores @ 3.0+ GHz - RAM: 32 GB minimum - Storage: 2 TB NVMe SSD - Network: 100 Mbps symmetric - Uptime: 99.5%+ target

Financial Requirements: - One-time: $800-$2,000 (hardware) - Monthly: $170-$450 (hosting, power, maintenance) - Expected revenue: $333+/month at $5M volume

3.5.2 Geographic Distribution

Target: 15 validators across 3+ continents

- North America: 5 validators
- Europe: 5 validators
- Asia: 3 validators
- Other regions: 2 validators

Constraints: - Maximum 3 validators per country - Maximum 2 validators per data center - Minimum 8 different countries

3.5.3 Recruitment Process

Week 1-4: Outreach (50+ candidates) - Bitcoin developer communities - Stacks ecosystem - Node operator networks - Conference networking

Week 5-8: Assessment (25 interviews) - Technical interviews (1 hour each) - Rust coding assessment - Infrastructure verification - Geographic screening

Week 9-12: Onboarding (15 selected) - Sign validator agreements - Hardware setup and verification - Software installation (testnet) - Communication channels - Testnet participation - Onboarding certification

3.5.4 Validator Agreement

Key terms: - 99.5%+ uptime requirement - 24/7 availability for critical incidents - Participate in >95% of signing rounds - Monthly validator calls - Quarterly DR drills - Fee-based compensation

3.5.5 Success Criteria

- ☐ 15 validators recruited by Month 10
- ☐ Geographic distribution achieved
- ☐ All validators pass technical assessment
- ☐ All complete onboarding training
- ☐ All have hardware ready for mainnet

---

4. Smart Contract Updates

4.1 Stacks Contract Changes for Phase 2

New Contract: `loan-protocol-v2.clar`

Key Additions: 1. Bitcoin address storage 2. Bitcoin deposit monitoring 3. Validator action requests 4. Cross-chain state tracking

```
;; ========================================
;; Phase 2 Extensions
;; ========================================

;; Bitcoin addresses for each loan
(define-map loan-bitcoin-addresses
  {loan-id: uint}
  {
    address: (string-ascii 62),         ;; bc1p... address
    deposit-txid: (optional (buff 32)), ;; Bitcoin txid
    deposit-confirmed: bool,
    confirmations: uint
  }
)

;; Validator action requests
(define-map action-requests
  {request-id: uint}
  {
    loan-id: uint,
    action-type: (string-ascii 20),     ;; "release", "claim", "refund"
    bitcoin-txid: (buff 32),            ;; UTXO to spend
    recipient-address: (string-ascii 62),
    amount-sats: uint,
    fee-sats: uint,
    requested-at-block: uint,
    status: (string-ascii 20),          ;; "pending", "signed", "confirmed"
    bitcoin-tx-result: (optional (buff 32))
  }
```

```
)

;; Validator registry
(define-map validators
  principal
  {
    validator-id: uint,
    public-key: (buff 33),
    status: (string-ascii 20),          ;; "active", "inactive"
    last-heartbeat: uint
  }
)

;; Action request counter
(define-data-var action-request-nonce uint u0)
```

4.2 Bitcoin Deposit Verification

```
;; Submit Bitcoin deposit proof
(define-public (submit-bitcoin-deposit-proof
    (loan-id uint)
    (txid (buff 32))
    (block-height uint)
    (merkle-proof (list 16 (buff 32)))
    (output-index uint)
    (output-amount uint))
  (let
    (
      (loan (unwrap! (map-get? loans {loan-id: loan-id}) ERR_LOAN_NOT_FOUND))
      (btc-address-info (unwrap! (map-get? loan-bitcoin-addresses {loan-id: loan-id}) ERR_NO
    )

    ;; Verify loan is awaiting deposit
    (asserts! (is-eq (get status loan) "awaiting-deposit") ERR_INVALID_STATUS)

    ;; Verify proof (simplified - actual implementation more complex)
    (asserts! (verify-merkle-proof txid block-height merkle-proof) ERR_INVALID_PROOF)

    ;; Verify amount matches
    (asserts! (>= output-amount (get collateral-amount loan)) ERR_INSUFFICIENT_AMOUNT)

    ;; Check confirmations
    (let ((current-height (get-bitcoin-block-height)))
      (let ((confirmations (- current-height block-height)))

        ;; Update address info
```

```
        (map-set loan-bitcoin-addresses
          {loan-id: loan-id}
          (merge btc-address-info {
            deposit-txid: (some txid),
            deposit-confirmed: (>= confirmations u3),
            confirmations: confirmations
          }))

        ;; If 3+ confirmations, activate loan
        (if (>= confirmations u3)
          (begin
            (map-set loans
              {loan-id: loan-id}
              (merge loan {status: "auction"}))
            (print {event: "deposit-confirmed", loan-id: loan-id, txid: txid})
          )
          (print {event: "deposit-pending", loan-id: loan-id, confirmations: confirmations})
        )

        (ok true)
      )
    )
  )
)

;; Helper: Verify merkle proof (simplified)
(define-read-only (verify-merkle-proof
    (txid (buff 32))
    (block-height uint)
    (proof (list 16 (buff 32))))
  ;; In production, this would:
  ;; 1. Get Bitcoin block header at block-height
  ;; 2. Verify merkle path from txid to merkle root
  ;; 3. Verify merkle root matches header
  ;; Simplified here for clarity
  true
)
```

4.3 Validator Action Requests

```
;; Request Bitcoin transaction (collateral release/claim)
(define-public (request-bitcoin-action
    (loan-id uint)
    (action-type (string-ascii 20))
    (recipient-address (string-ascii 62)))
  (let
```

```
(
  (loan (unwrap! (map-get? loans {loan-id: loan-id}) ERR_LOAN_NOT_FOUND))
  (btc-address-info (unwrap! (map-get? loan-bitcoin-addresses {loan-id: loan-id}) ERR_NO
  (request-id (+ (var-get action-request-nonce) u1))
  (deposit-txid (unwrap! (get deposit-txid btc-address-info) ERR_NO_DEPOSIT))
)

;; Verify caller is authorized
(asserts! (or
  (and (is-eq action-type "release") (is-loan-finalized loan-id))
  (and (is-eq action-type "claim") (is-loan-defaulted loan-id))
) ERR_UNAUTHORIZED)

;; Estimate fee (simplified - could be dynamic)
(let ((fee-sats u50000))  ;; ~$25 at $50K BTC

  ;; Create action request
  (map-set action-requests
    {request-id: request-id}
    {
      loan-id: loan-id,
      action-type: action-type,
      bitcoin-txid: deposit-txid,
      recipient-address: recipient-address,
      amount-sats: (get collateral-amount loan),
      fee-sats: fee-sats,
      requested-at-block: burn-block-height,
      status: "pending",
      bitcoin-tx-result: none
    })

  ;; Increment nonce
  (var-set action-request-nonce request-id)

  ;; Emit event for validators
  (print {
    event: "action-request",
    request-id: request-id,
    loan-id: loan-id,
    action: action-type,
    recipient: recipient-address,
    amount: (get collateral-amount loan),
    fee: fee-sats,
    utxo-txid: deposit-txid
  })
```

```
      (ok request-id)
    )
  )
)

;; Validators submit Bitcoin transaction confirmation
(define-public (confirm-bitcoin-action
    (request-id uint)
    (bitcoin-txid (buff 32))
    (block-height uint))
  (let
    (
      (request (unwrap! (map-get? action-requests {request-id: request-id}) ERR_REQUEST_NOT_
      (validator (unwrap! (map-get? validators tx-sender) ERR_NOT_VALIDATOR))
    )

    ;; Verify validator is active
    (asserts! (is-eq (get status validator) "active") ERR_VALIDATOR_INACTIVE)

    ;; Update request status
    (map-set action-requests
      {request-id: request-id}
      (merge request {
        status: "confirmed",
        bitcoin-tx-result: (some bitcoin-txid)
      }))

    ;; Update loan status
    (let ((loan-id (get loan-id request)))
      (let ((loan (unwrap! (map-get? loans {loan-id: loan-id}) ERR_LOAN_NOT_FOUND)))
        (map-set loans
          {loan-id: loan-id}
          (merge loan {
            status: (if (is-eq (get action-type request) "claim") "defaulted" "repaid")
          }))
      )
    )

    (print {
      event: "action-confirmed",
      request-id: request-id,
      bitcoin-txid: bitcoin-txid
    })

    (ok true)
  )
```

)

---

5. Bitcoin Transaction Construction

5.1 Transaction Builder

```rust
use bitcoin::{
    Transaction, TxIn, TxOut, OutPoint, Script, Witness,
    Address, Network, Amount
};
use bitcoin::blockdata::opcodes;
use bitcoin::secp256k1::{Secp256k1, PublicKey};

pub struct BitcoinTxBuilder {
    network: Network,
    fee_rate_sats_per_vbyte: u64,
}

impl BitcoinTxBuilder {
    pub fn new(network: Network) -> Self {
        Self {
            network,
            fee_rate_sats_per_vbyte: 50, // Conservative default
        }
    }

    /// Build collateral release transaction
    pub fn build_release_tx(
        &self,
        utxo_txid: [u8; 32],
        utxo_vout: u32,
        utxo_amount_sats: u64,
        recipient_address: &str,
        threshold_pubkey: &PublicKey,
    ) -> Result<Transaction, Error> {
        // Parse recipient address
        let recipient = Address::from_str(recipient_address)?
            .require_network(self.network)?;

        // Calculate fee
        let estimated_vsize = self.estimate_tx_vsize(1, 1); // 1 input, 1 output
        let fee_sats = estimated_vsize * self.fee_rate_sats_per_vbyte;
        let output_amount = utxo_amount_sats.saturating_sub(fee_sats);

        // Build transaction
```

```
    let tx = Transaction {
        version: 2,
        lock_time: bitcoin::PackedLockTime::ZERO,
        input: vec![
            TxIn {
                previous_output: OutPoint {
                    txid: bitcoin::Txid::from_slice(&utxo_txid)?,
                    vout: utxo_vout,
                },
                script_sig: Script::new(), // Empty for Taproot
                sequence: bitcoin::Sequence::ENABLE_RBF_NO_LOCKTIME,
                witness: Witness::new(), // Will be filled with signature
            }
        ],
        output: vec![
            TxOut {
                value: output_amount,
                script_pubkey: recipient.script_pubkey(),
            }
        ],
    };

    Ok(tx)
}

/// Estimate transaction virtual size
fn estimate_tx_vsize(&self, n_inputs: usize, n_outputs: usize) -> u64 {
    // Taproot key-path spend sizes (witness):
    // - 1 byte: witness stack items count
    // - 1 byte: signature length
    // - 64 bytes: Schnorr signature
    const TAPROOT_INPUT_WITNESS_SIZE: usize = 66;

    // Transaction overhead
    const TX_OVERHEAD: usize = 10;

    // Input size (excluding witness)
    const INPUT_SIZE: usize = 41; // outpoint (36) + script_sig (1) + sequence (4)

    // Output size
    const P2TR_OUTPUT_SIZE: usize = 43; // amount (8) + script (35)

    // Total weight units
    let base_weight = (TX_OVERHEAD + (INPUT_SIZE * n_inputs) + (P2TR_OUTPUT_SIZE * n_out
    let witness_weight = TAPROOT_INPUT_WITNESS_SIZE * n_inputs;
    let total_weight = base_weight + witness_weight;
```

```
        // Convert to vsize (weight / 4, rounded up)
        ((total_weight + 3) / 4) as u64
    }

    /// Attach threshold signature to transaction
    pub fn attach_signature(
        &self,
        mut tx: Transaction,
        signature: &[u8; 64], // Schnorr signature
    ) -> Transaction {
        // Create witness with signature
        let mut witness = Witness::new();
        witness.push(signature);

        // Attach to input
        tx.input[0].witness = witness;

        tx
    }

    /// Calculate transaction ID (txid)
    pub fn get_txid(tx: &Transaction) -> [u8; 32] {
        use bitcoin::hashes::{Hash, sha256d};
        let txid = tx.txid();
        txid.as_hash().into_inner()
    }
}
```

5.2 Fee Estimation

```
use bitcoincore_rpc::{RpcApi, Client};

pub struct FeeEstimator {
    rpc: Client,
}

impl FeeEstimator {
    pub fn new(rpc: Client) -> Self {
        Self { rpc }
    }

    /// Get current fee rate for target confirmation
    pub fn estimate_fee_rate(
        &self,
        target_blocks: u16,
```

```
    ) -> Result<u64, Error> {
        // Query Bitcoin Core for fee estimate
        let estimate = self.rpc.estimate_smart_fee(target_blocks, None)?;

        // Convert BTC/kB to sats/vbyte
        let btc_per_kb = estimate.fee_rate
            .ok_or(Error::FeeEstimationFailed)?;
        let sats_per_byte = (btc_per_kb * 100_000.0) as u64;

        // Add safety margin (10%)
        let sats_per_vbyte = sats_per_byte * 110 / 100;

        // Enforce minimum (1 sat/vbyte)
        Ok(sats_per_vbyte.max(1))
    }

    /// Get fee for specific transaction
    pub fn calculate_fee(
        &self,
        tx_vsize: u64,
        target_blocks: u16,
    ) -> Result<u64, Error> {
        let fee_rate = self.estimate_fee_rate(target_blocks)?;
        Ok(tx_vsize * fee_rate)
    }
}

// Example usage
let estimator = FeeEstimator::new(rpc_client);

// Target 2 block confirmation
let fee_rate = estimator.estimate_fee_rate(2)?;
println!("Current fee rate: {} sats/vbyte", fee_rate);

// Calculate fee for our transaction
let tx_fee = estimator.calculate_fee(141, 2)?;  // 141 vbytes typical
println!("Transaction fee: {} sats (${:.2})", tx_fee, tx_fee as f64 * btc_price / 100_000_0
```

5.3 Replace-By-Fee (RBF)

```
/// Bump transaction fee if stuck in mempool
pub fn create_rbf_transaction(
    original_tx: &Transaction,
    new_fee_rate_sats_per_vbyte: u64,
) -> Result<Transaction, Error> {
    // Clone original transaction
```

```
    let mut rbf_tx = original_tx.clone();

    // Calculate new fee
    let vsize = original_tx.vsize() as u64;
    let new_fee = vsize * new_fee_rate_sats_per_vbyte;

    // Calculate old fee
    let input_value = get_input_value(&original_tx)?;
    let output_value: u64 = original_tx.output.iter().map(|o| o.value).sum();
    let old_fee = input_value - output_value;

    // Ensure new fee is higher (RBF rule)
    if new_fee <= old_fee {
        return Err(Error::FeeNotHigherThanOriginal);
    }

    // Reduce output amount by fee difference
    let fee_increase = new_fee - old_fee;
    rbf_tx.output[0].value -= fee_increase;

    // Verify output is still above dust threshold
    const DUST_THRESHOLD: u64 = 546;
    if rbf_tx.output[0].value < DUST_THRESHOLD {
        return Err(Error::OutputBelowDust);
    }

    Ok(rbf_tx)
}
```

---

6. Cross-Chain State Management

6.1 State Synchronization

Challenge: Keep Stacks and Bitcoin states consistent despite different block times and reorg possibilities.

Solution: State machine with multiple confirmation levels.

```
#[derive(Debug, Clone, PartialEq)]
pub enum LoanState {
    // Phase 1: Loan creation (Stacks only)
    Created { stacks_block: u64 },

    // Phase 2: Bitcoin deposit (Cross-chain)
    AwaitingDeposit { bitcoin_address: String },
    DepositSeen { bitcoin_txid: String, confirmations: u8 },
```

```
    DepositConfirmed { bitcoin_txid: String },

    // Phase 3: Auction (Stacks only)
    AuctionActive { end_block: u64 },
    AuctionFinalized { winner: String },

    // Phase 4: Loan active (Cross-chain aware)
    Active { maturity_block: u64 },

    // Phase 5: Resolution (Cross-chain)
    RepaymentInitiated { stacks_block: u64 },
    CollateralReleasing { bitcoin_txid: Option<String> },
    Completed { bitcoin_txid: String },

    // Alternative: Default
    Defaulted { claimed_at: u64 },
}

pub struct CrossChainStateMachine {
    bitcoin_client: BitcoinClient,
    stacks_client: StacksClient,
}

impl CrossChainStateMachine {
    /// Monitor Bitcoin deposit
    pub async fn monitor_deposit(
        &self,
        loan_id: u64,
        bitcoin_address: &str,
    ) -> Result<LoanState, Error> {
        // Check if deposit exists
        let utxos = self.bitcoin_client.get_address_utxos(bitcoin_address).await?;

        if utxos.is_empty() {
            return Ok(LoanState::AwaitingDeposit {
                bitcoin_address: bitcoin_address.to_string()
            });
        }

        // Get first UTXO (should be only one)
        let utxo = &utxos[0];

        // Check confirmations
        let confirmations = self.bitcoin_client.get_confirmations(&utxo.txid).await?;

        if confirmations < 3 {
```

```
            Ok(LoanState::DepositSeen {
                bitcoin_txid: utxo.txid.clone(),
                confirmations: confirmations as u8,
            })
        } else {
            // Submit proof to Stacks
            self.submit_deposit_proof(loan_id, &utxo).await?;

            Ok(LoanState::DepositConfirmed {
                bitcoin_txid: utxo.txid.clone(),
            })
        }
    }

    /// Handle Bitcoin reorg
    pub async fn handle_reorg(
        &self,
        loan_id: u64,
        old_txid: &str,
    ) -> Result<LoanState, Error> {
        // Check if transaction still exists
        let tx_exists = self.bitcoin_client.check_tx_exists(old_txid).await?;

        if !tx_exists {
            // Reorg removed our transaction
            warn!("Deposit transaction {} was reorged out", old_txid);

            // Revert to awaiting deposit
            let loan = self.stacks_client.get_loan(loan_id).await?;
            let bitcoin_address = loan.bitcoin_address;

            Ok(LoanState::AwaitingDeposit { bitcoin_address })
        } else {
            // Transaction still exists, just different confirmations
            self.monitor_deposit(loan_id, &loan.bitcoin_address).await
        }
    }

    /// Monitor collateral release
    pub async fn monitor_collateral_release(
        &self,
        request_id: u64,
    ) -> Result<LoanState, Error> {
        // Check if Bitcoin transaction exists
        let request = self.stacks_client.get_action_request(request_id).await?;
```

```
        if let Some(bitcoin_txid) = request.bitcoin_tx_result {
            // Check confirmations
            let confirmations = self.bitcoin_client.get_confirmations(&bitcoin_txid).await?;

            if confirmations >= 1 {
                Ok(LoanState::Completed { bitcoin_txid })
            } else {
                Ok(LoanState::CollateralReleasing {
                    bitcoin_txid: Some(bitcoin_txid)
                })
            }
        } else {
            Ok(LoanState::CollateralReleasing {
                bitcoin_txid: None
            })
        }
    }
}
```

6.2 Reorg Handling

```
pub struct ReorgDetector {
    bitcoin_client: BitcoinClient,
    last_known_height: u64,
    last_known_hash: String,
}

impl ReorgDetector {
    pub async fn check_for_reorg(&mut self) -> Result<bool, Error> {
        let current_height = self.bitcoin_client.get_block_count().await?;

        if current_height < self.last_known_height {
            // Definitely a reorg
            warn!("Reorg detected: height decreased from {} to {}",
                  self.last_known_height, current_height);
            return Ok(true);
        }

        // Check if our last known block is still in the chain
        let block_hash = self.bitcoin_client
            .get_block_hash(self.last_known_height)
            .await?;

        if block_hash != self.last_known_hash {
            // Reorg happened
            warn!("Reorg detected: block hash changed at height {}",
```

```
                self.last_known_height);
            return Ok(true);
        }

        // Update tracking
        self.last_known_height = current_height;
        self.last_known_hash = self.bitcoin_client
            .get_block_hash(current_height)
            .await?;

        Ok(false)
    }

    pub async fn handle_reorg(&self, affected_loans: Vec<u64>) -> Result<(), Error> {
        for loan_id in affected_loans {
            // Re-verify deposit
            let loan = self.stacks_client.get_loan(loan_id).await?;

            if let Some(deposit_txid) = loan.deposit_txid {
                // Check if transaction still exists
                let exists = self.bitcoin_client.check_tx_exists(&deposit_txid).await?;

                if !exists {
                    // Transaction was reorged out
                    error!("Loan {} deposit was reorged out", loan_id);

                    // Notify contract (revert to awaiting deposit)
                    self.stacks_client.report_reorg(loan_id).await?;
                }
            }
        }

        Ok(())
    }
}
```

SECTION 6.3: Enhanced State Management (NEW)

Budget: $28,000
Timeline: Months 10-11 (2 months)
Deliverable: D2.4a

6.3.1 Overview

Enhanced state management ensures Bitcoin and Stacks blockchains remain consistent despite: - Bitcoin reorganizations (reorgs) - Network latency and asynchrony - Concurrent state updates - Node failures - Malicious attacks

Core Principle: Bitcoin blockchain is the source of truth.

6.3.2 Bitcoin Reorg Detection & Handling

Challenge: Bitcoin blockchain can reorganize when competing chains exist. Blocks can be orphaned.

Reorg Detection Algorithm:

```
struct ReorgDetector {
    bitcoin_client: BitcoinRpcClient,
    stacks_db: StacksDatabase,
    last_known_height: u64,
    last_known_hash: BlockHash,
    max_reorg_depth: u64, // 6 blocks
}

impl ReorgDetector {
    async fn check_for_reorg(&mut self) -> Result<Option<Reorg>, Error> {
        let current_height = self.bitcoin_client.get_block_count().await?;
        let current_hash = self.bitcoin_client.get_block_hash(current_height).await?;

        // Verify our last known block still in chain
        if current_height > self.last_known_height {
            let ancestor = self.bitcoin_client
                .get_block_hash(self.last_known_height)
                .await?;

            if ancestor != self.last_known_hash {
                // Reorg detected!
                return Ok(Some(self.trace_reorg(self.last_known_height).await?));
            }
        }

        Ok(None)
    }

    async fn trace_reorg(&self, start_height: u64) -> Result<Reorg, Error> {
        let mut orphaned_blocks = Vec::new();

        // Walk backwards to find common ancestor
        for depth in 0..self.max_reorg_depth {
```

```
            let height = start_height - depth;
            let our_hash = self.stacks_db.get_bitcoin_block_hash(height)?;
            let bitcoin_hash = self.bitcoin_client.get_block_hash(height).await?;

            if our_hash != bitcoin_hash {
                orphaned_blocks.push(OrphanedBlock {
                    height,
                    old_hash: our_hash,
                    new_hash: bitcoin_hash,
                });
            } else {
                break; // Found common ancestor
            }
        }

        Ok(Reorg {
            depth: orphaned_blocks.len() as u64,
            start_height,
            orphaned_blocks,
        })
    }
}
```

Reorg Handling Procedure:

```
impl StateManager {
    async fn handle_reorg(&mut self, reorg: Reorg) -> Result<(), Error> {
        info!("Handling reorg: depth={}", reorg.depth);

        // 1. Pause new transactions
        self.pause_transaction_processing();

        // 2. Identify affected loans
        let affected_loans = self.find_affected_loans(&reorg).await?;

        // 3. Roll back Stacks state
        self.rollback_stacks_state(reorg.start_height - reorg.depth).await?;

        // 4. Replay new chain
        for block in &reorg.orphaned_blocks {
            let new_block = self.bitcoin_client.get_block(&block.new_hash).await?;
            self.process_bitcoin_block(&new_block).await?;
        }

        // 5. Verify consistency
        self.verify_state_consistency().await?;
```

```
        // 6. Resume processing
        self.resume_transaction_processing();

        // 7. Notify users
        for loan_id in affected_loans {
            self.notify_loan_reorg(loan_id).await?;
        }

        Ok(())
    }
}
```

Guarantees: - Reorgs up to 6 blocks handled automatically - Affected loans identified - State consistency verified - No funds lost

6.3.3 Atomic State Updates

Challenge: Updates must be atomic across Bitcoin and Stacks.

Implementation:

```
pub struct AtomicUpdate {
    id: UpdateId,
    bitcoin_action: Option<BitcoinAction>,
    stacks_action: StacksAction,
    status: UpdateStatus,
    rollback_log: Vec<RollbackEntry>,
}

impl StateManager {
    async fn execute_atomic_update(&mut self, update: AtomicUpdate)
        -> Result<(), Error>
    {
        // Phase 1: Apply pending Stacks state
        self.apply_stacks_pending(&update).await?;

        // Phase 2: Execute Bitcoin action
        if let Some(action) = &update.bitcoin_action {
            match self.execute_bitcoin_action(action).await {
                Ok(txid) => self.mark_bitcoin_pending(&update, txid).await?,
                Err(e) => {
                    self.rollback_stacks(&update).await?;
                    return Err(e);
                }
            }

            // Phase 3: Wait for confirmations
```

```
            if !self.wait_for_confirmation(action.txid, 6).await? {
                self.rollback_stacks(&update).await?;
                return Err(Error::BitcoinTxFailed);
            }
        }

        // Phase 4: Apply final Stacks state
        self.apply_stacks_final(&update).await?;

        // Phase 5: Mark complete
        self.mark_update_complete(&update).await?;

        Ok(())
    }
}
```

Guarantees: - Both succeed or both rolled back - No partial updates - Crash recovery possible

6.3.4 Conflict Resolution

Strategy: Use Bitcoin timestamps as tie-breaker for concurrent updates.

```
impl StateManager {
    async fn resolve_conflicts(&mut self, updates: Vec<AtomicUpdate>)
        -> Result<Vec<AtomicUpdate>, Error>
    {
        let conflicts = self.detect_conflicts(&updates)?;

        for conflict in conflicts {
            // Select update with earliest Bitcoin timestamp
            let earliest = conflict.updates.iter()
                .min_by_key(|u| self.get_bitcoin_timestamp(u))
                .cloned();

            // Accept earliest, reject others
            // ...
        }

        Ok(resolved_updates)
    }
}
```

6.3.5 State Recovery

Strategy 1: Checkpoint Recovery (<5 minutes)

```rust
async fn recover_from_checkpoint(&mut self) -> Result<(), Error> {
    // Restore latest checkpoint
    let checkpoint = self.stacks_db.get_latest_checkpoint()?;
    self.stacks_db.restore_checkpoint(&checkpoint)?;

    // Replay blocks since checkpoint
    let current_height = self.bitcoin_client.get_block_count().await?;
    for height in (checkpoint.height + 1)..=current_height {
        let block = self.get_bitcoin_block(height).await?;
        self.process_bitcoin_block(&block).await?;
    }

    Ok(())
}
```

Strategy 2: Full Reconstruction (~2 hours) - Scan entire Bitcoin blockchain - Rebuild state from deposits/withdrawals - Verify all loans

6.3.6 State Verification

Automated checks (every block):

```rust
async fn verify_state_consistency(&self) -> Result<(), Error> {
    self.verify_deposits_match_loans().await?;
    self.verify_withdrawals_match_completions().await?;
    self.verify_utxo_set().await?;
    self.verify_state_hashes().await?;
    self.verify_no_double_spends().await?;
    Ok(())
}
```

Monitoring alerts: - State inconsistency: P1 (Critical) - Reorg detected: P2 (High) - Recovery required: P1 (Critical)

6.3.7 Success Criteria (D2.4a)

- ☐ Handle 6-block reorgs automatically
- ☐ Atomic updates: 100% success or rollback
- ☐ Checkpoint recovery: <5 minutes
- ☐ Full reconstruction: <2 hours
- ☐ Zero state inconsistencies in 30-day testnet

---

7. Validator Network Protocol

7.1 libp2p Network Configuration

```rust
use libp2p::{
    gossipsub, mdns, noise,
    swarm::{SwarmBuilder, SwarmEvent},
    tcp, yamux, PeerId, Transport,
};

pub struct ValidatorNetwork {
    swarm: Swarm<ValidatorBehaviour>,
    local_peer_id: PeerId,
}

#[derive(NetworkBehaviour)]
struct ValidatorBehaviour {
    gossipsub: gossipsub::Behaviour,
    mdns: mdns::tokio::Behaviour,
}

impl ValidatorNetwork {
    pub async fn new(config: &NetworkConfig) -> Result<Self, Error> {
        // Generate keypair for this validator
        let local_key = identity::Keypair::generate_ed25519();
        let local_peer_id = PeerId::from(local_key.public());

        info!("Validator peer ID: {}", local_peer_id);

        // Build transport
        let transport = tcp::tokio::Transport::new(tcp::Config::default().nodelay(true))
            .upgrade(upgrade::Version::V1)
            .authenticate(noise::Config::new(&local_key)?)
            .multiplex(yamux::Config::default())
            .boxed();

        // Configure Gossipsub
        let gossipsub_config = gossipsub::ConfigBuilder::default()
            .heartbeat_interval(Duration::from_secs(1))
            .validation_mode(gossipsub::ValidationMode::Strict)
            .build()?;

        let mut gossipsub = gossipsub::Behaviour::new(
            gossipsub::MessageAuthenticity::Signed(local_key.clone()),
            gossipsub_config,
        )?;
```

```rust
        // Subscribe to topics
        let nonce_topic = gossipsub::IdentTopic::new("validator/nonces");
        let signature_topic = gossipsub::IdentTopic::new("validator/signatures");
        gossipsub.subscribe(&nonce_topic)?;
        gossipsub.subscribe(&signature_topic)?;

        // Configure mDNS for local peer discovery
        let mdns = mdns::tokio::Behaviour::new(mdns::Config::default(), local_peer_id)?;

        // Build behaviour
        let behaviour = ValidatorBehaviour { gossipsub, mdns };

        // Build swarm
        let swarm = SwarmBuilder::with_tokio_executor(transport, behaviour, local_peer_id)
            .build();

        Ok(Self {
            swarm,
            local_peer_id,
        })
    }

    /// Broadcast nonce commitment
    pub fn broadcast_nonce_commitment(
        &mut self,
        signing_session: &str,
        commitment: &NonceCommitment,
    ) -> Result<(), Error> {
        let message = ValidatorMessage::NonceCommitment {
            session_id: signing_session.to_string(),
            validator_id: self.local_peer_id,
            commitment: commitment.clone(),
        };

        let topic = gossipsub::IdentTopic::new("validator/nonces");
        let serialized = serde_json::to_vec(&message)?;

        self.swarm
            .behaviour_mut()
            .gossipsub
            .publish(topic, serialized)?;

        Ok(())
    }
```

```rust
/// Broadcast partial signature
pub fn broadcast_partial_signature(
    &mut self,
    signing_session: &str,
    signature: &SignatureShare,
) -> Result<(), Error> {
    let message = ValidatorMessage::PartialSignature {
        session_id: signing_session.to_string(),
        validator_id: self.local_peer_id,
        signature: signature.clone(),
    };

    let topic = gossipsub::IdentTopic::new("validator/signatures");
    let serialized = serde_json::to_vec(&message)?;

    self.swarm
        .behaviour_mut()
        .gossipsub
        .publish(topic, serialized)?;

    Ok(())
}

/// Listen for messages
pub async fn next_event(&mut self) -> ValidatorNetworkEvent {
    loop {
        match self.swarm.select_next_some().await {
            SwarmEvent::Behaviour(BehaviourEvent::Gossipsub(
                gossipsub::Event::Message {
                    message,
                    ..
                }
            )) => {
                // Deserialize message
                if let Ok(msg) = serde_json::from_slice::<ValidatorMessage>(&message.dat
                    return ValidatorNetworkEvent::Message(msg);
                }
            }
            SwarmEvent::Behaviour(BehaviourEvent::Mdns(
                mdns::Event::Discovered(list)
            )) => {
                for (peer_id, addr) in list {
                    info!("Discovered peer: {} at {}", peer_id, addr);
                    self.swarm.dial(addr)?;
                }
            }
```

```
                _ => {}
            }
        }
    }
}

#[derive(Debug, Clone, Serialize, Deserialize)]
pub enum ValidatorMessage {
    NonceCommitment {
        session_id: String,
        validator_id: PeerId,
        commitment: NonceCommitment,
    },
    PartialSignature {
        session_id: String,
        validator_id: PeerId,
        signature: SignatureShare,
    },
    Heartbeat {
        validator_id: PeerId,
        timestamp: u64,
    },
}
```

7.2 Coordinator Service

```
pub struct Coordinator {
    network: ValidatorNetwork,
    active_sessions: HashMap<String, SigningSession>,
}

struct SigningSession {
    request_id: u64,
    message: [u8; 32],
    nonce_commitments: HashMap<PeerId, NonceCommitment>,
    partial_signatures: HashMap<PeerId, SignatureShare>,
    started_at: Instant,
}

impl Coordinator {
    /// Initiate signing session
    pub async fn initiate_signing(
        &mut self,
        request_id: u64,
        message: [u8; 32],
    ) -> Result<(), Error> {
```

```rust
    let session_id = format!("signing-{}", request_id);

    info!("Initiating signing session: {}", session_id);

    // Create session
    let session = SigningSession {
        request_id,
        message,
        nonce_commitments: HashMap::new(),
        partial_signatures: HashMap::new(),
        started_at: Instant::now(),
    };

    self.active_sessions.insert(session_id.clone(), session);

    // Broadcast signing request to all validators
    self.network.broadcast_signing_request(&session_id, message)?;

    Ok(())
}

/// Collect nonce commitments
pub async fn collect_nonce_commitments(
    &mut self,
    session_id: &str,
) -> Result<Vec<NonceCommitment>, Error> {
    let session = self.active_sessions.get_mut(session_id)
        .ok_or(Error::SessionNotFound)?;

    // Wait for at least 10 commitments (or timeout after 30 seconds)
    let timeout = Duration::from_secs(30);
    let start = Instant::now();

    while session.nonce_commitments.len() < 10 {
        if start.elapsed() > timeout {
            return Err(Error::InsufficientCommitments);
        }

        // Process network events
        if let ValidatorNetworkEvent::Message(
            ValidatorMessage::NonceCommitment {
                session_id: msg_session,
                validator_id,
                commitment,
            }
        ) = self.network.next_event().await {
```

```rust
                if msg_session == session_id {
                    info!("Received nonce commitment from {}", validator_id);
                    session.nonce_commitments.insert(validator_id, commitment);
                }
            }
        }

        Ok(session.nonce_commitments.values().cloned().collect())
    }

    /// Collect partial signatures
    pub async fn collect_partial_signatures(
        &mut self,
        session_id: &str,
    ) -> Result<Vec<SignatureShare>, Error> {
        let session = self.active_sessions.get_mut(session_id)
            .ok_or(Error::SessionNotFound)?;

        // Wait for at least 10 signatures
        let timeout = Duration::from_secs(30);
        let start = Instant::now();

        while session.partial_signatures.len() < 10 {
            if start.elapsed() > timeout {
                return Err(Error::InsufficientSignatures);
            }

            if let ValidatorNetworkEvent::Message(
                ValidatorMessage::PartialSignature {
                    session_id: msg_session,
                    validator_id,
                    signature,
                }
            ) = self.network.next_event().await {
                if msg_session == session_id {
                    info!("Received partial signature from {}", validator_id);
                    session.partial_signatures.insert(validator_id, signature);
                }
            }
        }

        Ok(session.partial_signatures.values().cloned().collect())
    }

    /// Aggregate signatures and create final Bitcoin transaction
    pub async fn finalize_signing(
```

```
        &mut self,
        session_id: &str,
        bitcoin_tx_builder: &BitcoinTxBuilder,
    ) -> Result<Transaction, Error> {
        let session = self.active_sessions.remove(session_id)
            .ok_or(Error::SessionNotFound)?;

        // Aggregate partial signatures
        let final_signature = frost::aggregate(
            &session.partial_signatures.values().cloned().collect::<Vec<_>>(),
            &session.nonce_commitments.values().cloned().collect::<Vec<_>>(),
        )?;

        info!("Successfully aggregated threshold signature");

        // Get Bitcoin transaction for this request
        let request = stacks_client.get_action_request(session.request_id).await?;

        let bitcoin_tx = bitcoin_tx_builder.build_release_tx(
            request.bitcoin_txid,
            0, // vout
            request.amount_sats,
            &request.recipient_address,
            &threshold_pubkey,
        )?;

        // Attach signature
        let signed_tx = bitcoin_tx_builder.attach_signature(bitcoin_tx, &final_signature);

        Ok(signed_tx)
    }
}
```

SECTION 7.3: Network Protocol Implementation (NEW)

Budget: $21,000
Timeline: Months 10-11 (2 months)
Deliverable: D2.4b

7.3.1 Overview

Validator network protocol enables 15 distributed validators to: - Communicate securely (libp2p + TLS) - Coordinate threshold signatures - Aggregate partial signatures - Maintain network health - Handle failures gracefully

Technology: - libp2p (P2P networking) - Gossip protocol (message propagation) - Protocol Buffers (serialization)

7.3.2 libp2p Architecture

```rust
use libp2p::{gossipsub, identity, noise, tcp, yamux};

pub struct ValidatorNetwork {
    swarm: Swarm<ValidatorBehaviour>,
    peer_id: PeerId,
    known_peers: HashMap<PeerId, ValidatorInfo>,
}

impl ValidatorNetwork {
    pub async fn new(config: NetworkConfig) -> Result<Self, Error> {
        // 1. Load or generate identity
        let keypair = load_or_generate_keypair(&config.identity_file)?;
        let peer_id = PeerId::from(keypair.public());

        // 2. Configure transport (TCP + encryption + multiplexing)
        let transport = tcp::async_io::Transport::default()
            .upgrade(libp2p::core::upgrade::Version::V1)
            .authenticate(noise::Config::new(&keypair)?)
            .multiplex(yamux::Config::default())
            .boxed();

        // 3. Configure behavior (gossipsub + identify + kademlia)
        let behaviour = ValidatorBehaviour {
            gossipsub: gossipsub::Behaviour::new(/* config */)?,
            identify: libp2p::identify::Behaviour::new(/* config */),
            kademlia: libp2p::kad::Behaviour::new(/* config */),
        };

        // 4. Create swarm
        let swarm = Swarm::with_async_std_executor(transport, behaviour, peer_id);

        Ok(Self { swarm, peer_id, known_peers: HashMap::new() })
    }

    pub async fn connect_to_peers(&mut self, peers: Vec<(PeerId, Multiaddr)>)
        -> Result<(), Error>
    {
        for (peer_id, addr) in peers {
            self.swarm.dial(addr)?;
            self.known_peers.insert(peer_id, ValidatorInfo { /* ... */ });
        }
```

```
        Ok(())
    }
}
```

Peer Discovery: - Bootstrap from seed nodes - Kademlia DHT for discovery - Wait for minimum 10/15 validators

Connection Management: - Max 50 connections - Min 10 connections - Keep-alive: 10s - Auto-reconnect on failure - NAT traversal enabled

7.3.3 Gossip Protocol

Message Types (Protocol Buffers):

```
message ValidatorMessage {
  oneof message_type {
    SignatureRequest signature_request = 1;
    PartialSignature partial_signature = 2;
    FinalSignature final_signature = 3;
    HealthCheck health_check = 4;
  }
}

message SignatureRequest {
  string request_id = 1;
  bytes transaction_hash = 2;
  uint64 timestamp = 3;
  uint32 required_signatures = 4;  // 10 of 15
  repeated string validator_ids = 5;
}

message PartialSignature {
  string request_id = 1;
  string validator_id = 2;
  bytes signature_share = 3;
  uint64 timestamp = 4;
}
```

Gossip Implementation:

```
impl ValidatorNetwork {
    pub async fn broadcast_signature_request(&mut self, request: SignatureRequest)
        -> Result<(), Error>
    {
        let message = ValidatorMessage {
            message_type: Some(MessageType::SignatureRequest(request.clone())),
        };
```

```
        let topic = gossipsub::IdentTopic::new("validator-signatures");
        self.swarm.behaviour_mut().gossipsub.publish(topic, message.encode())?;

        self.pending_requests.insert(request.request_id.clone(), /* ... */);
        Ok(())
    }

    async fn handle_signature_request(&mut self, request: SignatureRequest)
        -> Result<(), Error>
    {
        // Verify request
        if !self.verify_signature_request(&request).await? {
            return Ok(());
        }

        // Generate partial signature
        let partial = self.sign_partial(&request).await?;

        // Broadcast partial signature
        self.broadcast_partial_signature(partial).await?;

        Ok(())
    }

    async fn handle_partial_signature(&mut self, partial: PartialSignature)
        -> Result<(), Error>
    {
        let pending = self.pending_requests.get_mut(&partial.request_id)?;

        // Verify and store
        if self.verify_partial_signature(&partial, &pending.request).await? {
            pending.partial_signatures.push(partial);

            // Aggregate if enough signatures
            if pending.partial_signatures.len() >= pending.request.required_signatures {
                self.aggregate_signatures(&partial.request_id).await?;
            }
        }

        Ok(())
    }
}
```

Optimizations: - Deduplication (prevent message flooding) - Priority queuing (urgent requests first) - Seen messages cache (5 min TTL)

7.3.4 Signature Aggregation

```rust
impl ValidatorNetwork {
    async fn aggregate_signatures(&mut self, request_id: &str)
        -> Result<FinalSignature, Error>
    {
        let pending = self.pending_requests.get(request_id)?;

        // 1. Select best M signatures (10 of available)
        let selected = self.select_partial_signatures(
            &pending.partial_signatures,
            pending.request.required_signatures as usize
        )?;

        // 2. Verify all selected
        for partial in &selected {
            self.verify_partial_signature(partial, &pending.request).await?;
        }

        // 3. Aggregate using threshold crypto
        let aggregated = self.crypto.aggregate_partial_signatures(
            &pending.request.transaction_hash,
            &selected.iter().map(|p| p.signature_share.clone()).collect()
        )?;

        // 4. Verify final signature
        self.crypto.verify_signature(
            &aggregated,
            &pending.request.transaction_hash,
            &self.combined_public_key
        )?;

        // 5. Broadcast final signature
        let final_sig = FinalSignature {
            request_id: request_id.to_string(),
            aggregated_signature: aggregated,
            contributing_validators: selected.iter()
                .map(|p| p.validator_id.clone())
                .collect(),
            timestamp: current_timestamp(),
        };

        self.broadcast_final_signature(&final_sig).await?;

        Ok(final_sig)
    }
```

```
    fn select_partial_signatures(
        &self,
        partials: &[PartialSignature],
        required: usize
    ) -> Result<Vec<PartialSignature>, Error> {
        // Select validators with best performance
        let mut scored: Vec<_> = partials.iter()
            .map(|p| (p.clone(), self.validator_performance_score(&p.validator_id)))
            .collect();

        scored.sort_by(|a, b| b.1.partial_cmp(&a.1).unwrap());

        Ok(scored.into_iter().take(required).map(|(p, _)| p).collect())
    }
}
```

Guarantees: - Minimum 10 signatures required - All partials verified before aggregation - Final signature verified - Best validators prioritized - Completes in <10 seconds

7.3.5 Network Topology

Full Mesh (default): - Each validator connects to all 14 others - 105 total connections (15 choose 2) - Maximum redundancy - Lowest latency

Geographic Optimization:

```
fn optimize_topology_for_geography(&mut self) -> Result<(), Error> {
    // Connect to at least one validator per region
    for (region, peers) in self.validators_by_region() {
        let closest = peers.iter().min_by_key(|p| self.estimate_latency(p))?;
        self.ensure_connection(*closest)?;
    }

    // Fill remaining with lowest latency peers
    let remaining = 14 - self.swarm.connected_peers().count();
    for peer in self.get_lowest_latency_peers(remaining) {
        self.ensure_connection(peer)?;
    }

    Ok(())
}
```

Latency Monitoring: - Ping/pong every 60 seconds - Track p50, p95, p99 - Alert if >2s p95 - Auto-optimize connections

7.3.6 Network Resilience

Validator Failure Detection:

```rust
async fn detect_failed_validators(&mut self) {
    let mut failed = Vec::new();

    for (peer_id, info) in &self.known_peers {
        // Check connection
        if !self.swarm.is_connected(peer_id) {
            failed.push(*peer_id);
        }

        // Check last seen (5 min threshold)
        if info.last_seen.elapsed() > Duration::from_secs(300) {
            failed.push(*peer_id);
        }

        // Check health checks
        if !self.received_recent_health_check(peer_id) {
            failed.push(*peer_id);
        }
    }

    self.failed_validators = failed;

    if self.failed_validators.len() > 5 {
        error!("CRITICAL: {} validators failed!", self.failed_validators.len());
        self.send_critical_alert().await;
    }
}
```

Network Partition Recovery:

```rust
async fn detect_network_partition(&mut self) -> Option<NetworkPartition> {
    let connected = self.swarm.connected_peers().count();
    let total = self.known_peers.len();

    // Partition if <67% reachable
    if connected < (total * 2 / 3) {
        warn!("Network partition: {}/{} reachable", connected, total);
        return Some(NetworkPartition { /* ... */ });
    }

    None
}
```

```rust
async fn recover_from_partition(&mut self, partition: NetworkPartition)
    -> Result<(), Error>
{
    // Try reconnecting
    for peer_id in &partition.unreachable {
        if let Some(info) = self.known_peers.get(peer_id) {
            let _ = self.swarm.dial(info.addr.clone());
        }
    }

    // Try backup addresses
    for peer_id in &partition.unreachable {
        if let Some(backups) = self.config.backup_addresses.get(peer_id) {
            for addr in backups {
                let _ = self.swarm.dial(addr.clone());
            }
        }
    }

    // Wait for recovery (2 min timeout)
    tokio::time::timeout(
        Duration::from_secs(120),
        self.wait_for_partition_recovery()
    ).await?;

    Ok(())
}
```

Message Retry: - Max 3 retries - Exponential backoff (2^n seconds) - Timeout after 3 attempts

7.3.7 Performance Monitoring

Metrics (Prometheus):

```rust
pub struct NetworkMetrics {
    messages_sent: Counter,
    messages_received: Counter,
    message_latency: Histogram,
    active_connections: Gauge,
    signature_requests_sent: Counter,
    partial_signatures_received: Counter,
    signature_aggregation_time: Histogram,
    failed_validators: Gauge,
    peer_latencies: GaugeVec,
}
```

```rust
fn export_metrics(&self) -> String {
    format!(
        "validator_messages_sent {}\n\
         validator_messages_received {}\n\
         validator_active_connections {}\n\
         validator_signature_requests {}\n",
        self.metrics.messages_sent.get(),
        self.metrics.messages_received.get(),
        self.swarm.connected_peers().count(),
        self.metrics.signature_requests_sent.get()
    )
}
```

7.3.8 Testing Scenarios

1. **Normal**: 15 validators, <2s latency
2. **Single failure**: 14 validators, continues
3. **5 failures**: 10 validators, threshold met
4. **6 failures**: Cannot sign, alerts
5. **Partition**: 8-7 split, larger continues
6. **High latency**: 5s to 5 validators
7. **Message loss**: 20% packet loss
8. **Byzantine**: 1-3 malicious validators

7.3.9 Success Criteria (D2.4b)

- ☐ All 15 validators connected
- ☐ Gossip <5 seconds to all peers
- ☐ Aggregation <10 seconds
- ☐ Handles 10+ requests/minute
- ☐ Resilient to 5/15 failures
- ☐ Partition recovery <2 minutes
- ☐ Latency p95: <2 seconds
- ☐ Message delivery: >99%

[Document continues with sections 8-13...]

Due to length, I'll continue in the next message. Would you like me to complete the remaining sections (8. Security Architecture through 13. Disaster Recovery)?

8. Security Architecture

8.1 Threat Model

Assets to Protect: 1. Validator key shares (most critical) 2. Bitcoin in threshold addresses 3. Validator network integrity 4. Cross-chain state consistency

Threat Actors: - External attackers (steal Bitcoin) - Malicious validators (collude or sabotage) - Network attackers (DoS, eclipse) - State inconsistency exploits

Attack Scenarios & Mitigations:

Attack	Likelihood	Impact	Mitigation
Key extraction from single validator	Low	Critical if 10+ compromised	HSMs, secure enclaves, monitoring
Validator collusion (10+)	Very Low	Critical	Geographic distribution, economic incentives, reputation
Network partition	Medium	High	Multiple network paths, partition detection
Double-signing	Low	Medium	Nonce tracking, replay prevention
Eclipse attack on validator	Medium	Medium	Multiple bootstrap peers, peer diversity
Stacks contract exploit	Low	Critical	Audit, formal verification, bug bounty
Bitcoin reorg	Low	Medium	Wait for 3+ confirmations, reorg detection
Coordinator compromise	Medium	Medium	Coordinator is not trusted (verify sigs), multiple coordinators possible

8.2 Key Share Security

HSM Requirements:

```
Hardware Security Module (HSM) must provide:
- FIPS 140-2 Level 3 or higher
```

```
- Tamper detection and response
- Secure key storage (keys never leave HSM)
- Cryptographic acceleration
- Audit logging
- Backup and recovery under wrap

Recommended: YubiHSM 2, AWS CloudHSM, Thales Luna
```

Key Generation Security:

```
// Never construct full private key in any location
// Each validator only ever has their share

// BAD - DO NOT DO THIS:
let full_private_key = combine_shares(shares); //  NEVER

// GOOD - Validator only uses their share:
let my_share = load_from_hsm()?;
let partial_sig = sign_with_share(&my_share, message)?;
drop(my_share); // Immediately drop after use
```

Key Share Backup:

```
Backup Strategy:
1. Export wrapped key from HSM (encrypted)
2. Split wrapped key using Shamir Secret Sharing (3-of-5)
3. Distribute shares to:
   - Validator operator (encrypted)
   - Secure cloud storage (2 different providers)
   - Hardware backup (offline, different location)
   - Escrow service (for emergency recovery)

Recovery requires 3 of 5 shares
Test recovery procedure quarterly
```

8.3 Enhanced Security Audit (D2.6)

Budget: $82,000 (increased from $52,000)
Timeline: Months 12-13 (2 months)

8.3.1 Why $82K vs $52K? Phase 2 complexity requires specialized expertise:

+$15K: Threshold Cryptography Specialist - FROST/MuSig2 are cutting-edge - Few auditors have deep threshold crypto expertise - Critical to protocol security

+$8K: Distributed Systems Audit - 15-validator network = new attack surfaces - P2P security (libp2p, gossip) - Byzantine fault scenarios

+$5K: Economic Game Theory - Validator incentive analysis - Collusion resistance - 51% attack scenarios

+$2K: Extended Time - ~25,000 LOC (vs ~15,000 LOC original) - More complex integration testing

8.3.2 Audit Breakdown Phase 1: Threshold Crypto Audit ($30K, 2 weeks)

Specialized firm (Trail of Bits or Least Authority):

Scope: - [ ] FROST/MuSig2 implementation correctness - [ ] DKG ceremony security - Key generation randomness - Verifiable secret sharing - Malicious participant detection - [ ] Partial signature generation - Key share protection - Signature uniqueness - Nonce handling - [ ] Signature aggregation - Partial sig verification - Aggregation correctness - Final sig validity - [ ] Key storage & encryption - HSM integration - File-based security - Backup procedures - [ ] Taproot compatibility - P2TR address generation - Schnorr format - Bitcoin consensus rules

Deliverables: - Threshold crypto audit report - Cryptographic primitive analysis - Implementation recommendations - Test vectors

Success: - Zero critical vulnerabilities - All ops formally verified - Cryptographer sign-off

---

Phase 2: Validator Network Audit ($22K, 2 weeks)

Distributed systems experts:

Scope: - [ ] libp2p network security - Peer authentication - TLS encryption - Sybil resistance - [ ] Gossip protocol vulnerabilities - Message flooding - Eclipse attacks - Network partitioning - [ ] Signature request protocol - Replay prevention - Spoofing prevention - Priority manipulation - [ ] Consensus mechanisms - Byzantine fault tolerance - 10-of-15 enforcement - Validator exclusion - [ ] State synchronization - Cross-chain consistency - Reorg handling - Atomic updates - [ ] Validator authentication - Identity verification - Key rotation - Access control

Deliverables: - Network security report - Attack vector analysis - Penetration test results - Mitigation recommendations

Success: - Resistant to 5 malicious validators - No eclipse/Sybil attacks - Partition recovery verified

---

Phase 3: Bridge Logic Audit ($15K, 1 week)

Smart contract & cross-chain:

Scope: - [ ] Stacks Bitcoin bridge - Deposit verification - Withdrawal authorization - UTXO tracking - [ ] Bitcoin transaction construction - Fee estimation security - RBF manipulation resistance - Transaction malleability - [ ] Cross-chain state management - State consistency - Reorg handling - Rollback procedures - [ ] Smart contracts (Clarity) - Reentrancy protection - Access control - Integer overflow/underflow

Deliverables: - Bridge security report - Smart contract audit - Cross-chain attack vectors - Remediation plan

Success: - No bridge exploits - Funds cannot be locked/stolen - State consistency guaranteed

---

Phase 4: Economic Attack Vectors ($10K, 1 week)

Game theory analysis:

Scope: - [ ] Validator incentive analysis - Profit from honesty - Attack costs - Collusion resistance - [ ] 51% attack scenarios - Cost to acquire 8+ validators - Damage possible - Detection/mitigation - [ ] Griefing attacks - Denial of service - Validator exclusion - Transaction censorship - [ ] Fee manipulation - Bitcoin fee market gaming - Stablecoin price manipulation - Interest rate manipulation - [ ] Economic sustainability - Validator profitability - Long-term incentives - Protocol fee adequacy

Deliverables: - Economic security report - Game theory analysis - Attack cost calculations - Incentive recommendations

Success: - Attack costs > potential gains - Validator incentives aligned - Protocol sustainable

---

Phase 5: Follow-up Re-audit ($5K, 1 week)

After fixes:

Scope: - [ ] Verify critical findings fixed - [ ] Re-test high severity - [ ] Regression testing - [ ] Final sign-off

Deliverables: - Re-audit confirmation - Verification of fixes - Final assessment - Public disclosure

Success: - All critical resolved - All high resolved - Medium: resolved or accepted - Audit firm sign-off

---

8.3.3 Audit Firm Selection Preferred (priority order):

1. **Trail of Bits** (threshold crypto specialists)

- Ethereum 2.0, Zcash audits
- Cost: $80K-$90K

2. **Least Authority** (distributed systems)
   - Founded by Zooko Wilcox
   - Filecoin, Tezos audits
   - Cost: $70K-$85K
3. **CoinFabrik** (Stacks/Clarity)
   - Stacks ecosystem expertise
   - Cost-effective
   - Cost: $60K-$75K
4. **Halborn** (comprehensive blockchain)
   - Full-stack audits
   - Cost: $75K-$90K

Process: 1. RFP to all 4 firms (Month 10) 2. Proposals reviewed (Month 11) 3. Select based on: - Threshold crypto expertise - Cost ($82K) - Availability (Month 12-13) - Reputation 4. Contract signed, audit begins (Month 12)

8.3.4 Audit Timeline

```
Month 12:
  Week 1-2: Threshold crypto (Phase 1)
  Week 3-4: Validator network (Phase 2)

Month 13:
  Week 1: Bridge + economic (Phase 3-4)
  Week 2: Initial report delivered
  Week 3: Team addresses findings
  Week 4: Re-audit (Phase 5), final report
```

8.3.5 Acceptance Criteria Before mainnet:

Critical: ZERO (must fix, re-audit)
High: ALL RESOLVED (must fix or mitigate)
Medium: ADDRESSED (fixed, mitigated, or accepted)
Low: DOCUMENTED (can backlog)

Final Sign-off: - [ ] Audit firm written approval - [ ] Final report published publicly - [ ] No outstanding critical/high - [ ] Team acceptance

8.3.6 Post-Audit Bug Bounty **Launch**: Month 14, Week 2
Budget: $50K+ rewards

Rewards: - Critical (funds at risk): $10K-$50K - High (system compromise): $5K-$15K - Medium (degraded service): $1K-$5K - Low (minor): $500-$1K

Scope: - Threshold crypto - Validator network - Smart contracts - Bridge logic - State management - Key storage

Exclusions: - Known issues from audit - Social engineering - Physical attacks - DDoS

Process: - Submit to security@protocol.example - Responsible disclosure - 90-day embargo

Target: <5 valid submissions in 3 months

8.X Security Investment Summary

Total Security: $82K (D2.6)

Breakdown: - Threshold crypto: $30K (37%) - Validator network: $22K (27%) - Bridge logic: $15K (18%) - Economic analysis: $10K (12%) - Re-audit: $5K (6%)

Additional: - Bug bounty: $50K+ (post-launch) - Monitoring: D2.10 - Incident response: D2.11 - Disaster recovery: D2.11

Total Investment: $132K+ (29% of $463K)

Industry-leading security investment.

8.4 Double-Signing Prevention

```rust
pub struct NonceTracker {
    used_nonces: HashSet<([u8; 32], u64)>, // (message, session_id)
}

impl NonceTracker {
    /// Ensure we never sign same message twice with different nonce
    pub fn check_and_record(
        &mut self,
        message: &[u8; 32],
        session_id: u64,
    ) -> Result<(), Error> {
        let key = (*message, session_id);

        if self.used_nonces.contains(&key) {
            error!("Attempted double-signing detected!");
            return Err(Error::DoubleSigningAttempt);
        }

        self.used_nonces.insert(key);
        Ok(())
    }
```

```rust
    /// Clean old nonces (after confirmed on Bitcoin)
    pub fn cleanup_old_nonces(&mut self, cutoff: u64) {
        self.used_nonces.retain(|(_, session_id)| *session_id > cutoff);
    }
}
```

---

9. API Specifications

9.1 Validator REST API

Base URL: `http://validator-host:8080/api/v1`

Endpoints:

```
GET /health
Response: {"status": "healthy", "bitcoin_synced": true, "stacks_synced": true}

GET /status
Response: {
  "validator_id": 1,
  "peer_count": 14,
  "last_signature": "2026-01-12T10:30:00Z",
  "uptime_seconds": 864000
}

GET /key_share/public
Response: {
  "public_key_share": "02abc123...",
  "combined_public_key": "03def456..."
}

POST /signing/request
Body: {
  "session_id": "signing-123",
  "message": "abc123...",
  "participants": [1, 2, 3, ...]
}
Response: {"status": "initiated"}

GET /signing/session/:session_id
Response: {
  "session_id": "signing-123",
  "status": "collecting_signatures",
  "nonce_commitments": 12,
  "partial_signatures": 10,
```

```
  "progress": "67%"
}
```

9.2 Coordinator REST API

```
POST /api/v1/action/initiate
Body: {
  "request_id": 42,
  "action_type": "release",
  "bitcoin_tx_params": {...}
}
Response: {
  "session_id": "signing-42",
  "status": "initiated"
}

GET /api/v1/action/status/:request_id
Response: {
  "request_id": 42,
  "session_id": "signing-42",
  "status": "signed",
  "bitcoin_txid": "abc123...",
  "confirmations": 2
}

GET /api/v1/validators
Response: {
  "validators": [
    {"id": 1, "status": "active", "last_seen": "2s ago"},
    {"id": 2, "status": "active", "last_seen": "1s ago"},
    ...
  ],
  "online": 14,
  "threshold": 10
}
```

9.3 WebSocket Events

```
// Connect to validator WebSocket
const ws = new WebSocket('ws://validator-host:8080/ws');

// Subscribe to events
ws.send(JSON.stringify({
  type: 'subscribe',
  topics: ['signatures', 'deposits', 'network']
}));
```

```javascript
// Listen for events
ws.onmessage = (event) => {
  const data = JSON.parse(event.data);

  switch (data.type) {
    case 'signature_started':
      console.log(`Signing session ${data.session_id} started`);
      break;

    case 'signature_completed':
      console.log(`Bitcoin tx ${data.txid} signed`);
      break;

    case 'deposit_detected':
      console.log(`Deposit to loan ${data.loan_id}: ${data.txid}`);
      break;

    case 'network_event':
      console.log(`Peer ${data.peer_id}: ${data.status}`);
      break;
  }
};
```

---

10. Testing Strategy

10.1 Unit Tests

Threshold Cryptography:

```rust
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_dkg_ceremony() {
        // Simulate 15 validators
        let mut validators = vec![];
        for i in 0..15 {
            validators.push(Validator::new(i));
        }

        // Round 1: Commitments
        let commitments: Vec<_> = validators
            .iter_mut()
```

```
        .map(|v| v.dkg_round1())
        .collect();

    // Round 2: Shares
    let shares = validators[0].dkg_round2(&commitments);

    // Each validator verifies shares
    for validator in &validators {
        assert!(validator.verify_shares(&shares));
    }

    // Derive combined public key
    let pub_keys: Vec<_> = validators
        .iter()
        .map(|v| v.get_public_key_share())
        .collect();

    let combined_key = combine_public_keys(&pub_keys);

    // Test signing with 10 validators
    let message = b"test message";
    let signing_validators = &validators[0..10];

    let partial_sigs: Vec<_> = signing_validators
        .iter()
        .map(|v| v.create_partial_signature(message))
        .collect();

    let final_sig = aggregate_signatures(&partial_sigs);

    // Verify signature
    assert!(verify_signature(&final_sig, message, &combined_key));
}

#[test]
fn test_insufficient_signatures() {
    // Only 9 signatures (need 10)
    let partial_sigs = vec![/* 9 signatures */];

    let result = aggregate_signatures(&partial_sigs);
    assert!(result.is_err());
}

#[test]
fn test_malicious_signature() {
    // One signature is invalid
```

```
        let mut partial_sigs = vec![/* 9 valid signatures */];
        partial_sigs.push(create_invalid_signature());

        let result = aggregate_signatures(&partial_sigs);
        assert!(result.is_err());
    }
}
```

Bitcoin Transaction Construction:

```
#[test]
fn test_release_transaction() {
    let builder = BitcoinTxBuilder::new(Network::Testnet);

    let utxo_txid = [0u8; 32];
    let utxo_vout = 0;
    let utxo_amount = 150_000_000; // 1.5 BTC
    let recipient = "tb1q...";
    let pubkey = PublicKey::from_slice(&[0x02; 33]).unwrap();

    let tx = builder.build_release_tx(
        utxo_txid,
        utxo_vout,
        utxo_amount,
        recipient,
        &pubkey,
    ).unwrap();

    // Verify transaction structure
    assert_eq!(tx.input.len(), 1);
    assert_eq!(tx.output.len(), 1);

    // Verify fee is reasonable
    let output_amount = tx.output[0].value;
    let fee = utxo_amount - output_amount;
    assert!(fee > 1000); // At least 1000 sats
    assert!(fee < 100_000); // But not excessive
}
```

10.2 Integration Tests

Full Signing Flow:

```
#[tokio::test]
async fn test_complete_signing_flow() {
    // Setup 15 validators
    let validators = setup_test_validators(15).await;
```

```rust
    let coordinator = Coordinator::new();

    // Create signing request
    let message = create_test_message();
    let session_id = coordinator
        .initiate_signing(1, message)
        .await
        .unwrap();

    // Validators respond with nonce commitments
    for validator in &validators {
        validator
            .respond_to_signing_request(&session_id, &message)
            .await
            .unwrap();
    }

    // Coordinator collects commitments
    let commitments = coordinator
        .collect_nonce_commitments(&session_id)
        .await
        .unwrap();

    assert!(commitments.len() >= 10);

    // Validators create partial signatures
    for validator in &validators {
        validator
            .create_partial_signature(&session_id, &commitments)
            .await
            .unwrap();
    }

    // Coordinator aggregates
    let final_sig = coordinator
        .finalize_signing(&session_id)
        .await
        .unwrap();

    // Verify final signature
    assert!(verify_threshold_signature(&final_sig, &message));
}
```

Cross-Chain State Sync:

```rust
#[tokio::test]
async fn test_deposit_confirmation() {
```

```rust
    let bitcoin_client = setup_bitcoin_regtest().await;
    let stacks_client = setup_stacks_mocknet().await;

    // Create loan on Stacks
    let loan_id = stacks_client.create_loan().await.unwrap();

    // Get Bitcoin address
    let btc_address = stacks_client.get_loan_address(loan_id).await.unwrap();

    // Send Bitcoin
    let txid = bitcoin_client
        .send_to_address(&btc_address, 150_000_000)
        .await
        .unwrap();

    // Mine 3 blocks
    bitcoin_client.generate_blocks(3).await.unwrap();

    // Submit proof to Stacks
    let proof = bitcoin_client.get_tx_proof(&txid).await.unwrap();
    stacks_client
        .submit_deposit_proof(loan_id, proof)
        .await
        .unwrap();

    // Verify loan activated
    let loan = stacks_client.get_loan(loan_id).await.unwrap();
    assert_eq!(loan.status, "auction");
}
```

10.3 Chaos Testing

```rust
/// Test validator failures during signing
#[tokio::test]
async fn test_validator_failure_during_signing() {
    let mut validators = setup_test_validators(15).await;
    let coordinator = Coordinator::new();

    // Start signing
    let session_id = coordinator.initiate_signing(1, message).await.unwrap();

    // 12 validators respond
    for i in 0..12 {
        validators[i].respond(&session_id).await.unwrap();
    }
```

```rust
    // 3 validators fail to respond (simulated)
    // But we have 12 > 10, so signing should succeed

    let result = coordinator.finalize_signing(&session_id).await;
    assert!(result.is_ok());
}

/// Test network partition
#[tokio::test]
async fn test_network_partition() {
    let validators = setup_test_validators(15).await;

    // Create network partition: 8 vs 7 validators
    let (group_a, group_b) = validators.split_at(8);

    // Disconnect groups
    partition_network(group_a, group_b).await;

    // Larger group (8) should continue
    let result_a = group_a[0].initiate_signing(message).await;
    assert!(result_a.is_err()); // 8 < 10, should fail

    // Reconnect
    reconnect_network(group_a, group_b).await;

    // Now all 15 can sign
    let result = validators[0].initiate_signing(message).await;
    assert!(result.is_ok());
}
```

10.4 DKG Ceremony Testing (D2.3a)

Objective: Ensure DKG success under various conditions

Testnet Rehearsals (Min 2):

Rehearsal 1: Ideal

```yaml
scenario: "All 15 validators, perfect network"
participants: 15
network: "normal"
duration: "<10 minutes"
success:
  - All generate key shares
  - Public key Taproot-compatible
  - Test tx signs successfully
```

```
  - Zero errors
```

Rehearsal 2: Failures

```
scenario: "3 validators fail during ceremony"
participants: 15 (3 fail)
network: "normal"
duration: "<15 minutes"
expected: "DKG excludes failed, continues with 12"
success:
  - Completes with 12 validators
  - Failed identified and excluded
  - Public key valid
  - 10-of-12 achievable
```

Rehearsal 3: Byzantine (optional)

```
scenario: "2 validators malicious"
participants: 15 (2 byzantine)
behavior: "invalid commitments, bad shares"
expected: "Byzantine detected and excluded"
success:
  - Malicious identified
  - Completes with 13 honest
  - No key compromise
```

Checklist: - [ ] 15 validators complete rehearsal 1 - [ ] Ceremony <10 minutes - [ ] All generate key shares - [ ] Keys encrypted and stored - [ ] Public key verified on testnet - [ ] Test tx signed (10-of-15) - [ ] Test tx confirmed on testnet - [ ] Rehearsal 2 completed - [ ] Audit logs reviewed - [ ] Procedures documented

Mainnet Readiness: - [ ] 2+ testnet rehearsals - [ ] All validators trained - [ ] Coordinator ready (D2.3a) - [ ] Rollback procedures tested - [ ] Communication channels tested - [ ] Backup date scheduled

---

10.5 Disaster Recovery Testing (D2.11)

Budget: $12K
Timeline: Month 13

Objective: Verify DR procedures under realistic failures

Test 1: Key Recovery (Min 3 tests)

```
scenario: "3 validators lose key shares"
affected: 3
failure: "hardware failure, keys lost"
procedure:
  1. Simulate loss on 3 validators
```

```
  2. Retrieve encrypted backups
  3. Decrypt with master key
  4. Restore to validators
  5. Verify signing works
  6. Sign test transaction
duration: "<2 hours"
success:
  - All 3 recovered
  - Recovered validators sign
  - Test tx signed (10-of-15)
  - Zero data loss
```

Test 2: Network Partition (Min 5 tests)

```
scenario: "10-5 partition"
partition: "10 in A, 5 in B"
procedure:
  1. Introduce partition (firewall)
  2. Partition A can sign (10-of-10)
  3. Partition B detects below threshold
  4. Remove partition
  5. Network heals automatically
duration: "<5 min partition, <2 min recovery"
success:
  - Larger partition signs
  - Smaller detects problem
  - Recovers automatically
  - No signatures lost
```

Test 3: Coordinator Failover (Min 2 tests)

```
scenario: "Primary coordinator crashes"
failure: "kill -9"
procedure:
  1. Initiate signature via primary
  2. Kill primary mid-request
  3. Backup detects failure
  4. Backup takes over
  5. Request completes
duration: "<30s failover"
success:
  - Backup assumes role
  - Request completes
  - Zero downtime
  - All validators aware
```

Test 4: Bitcoin Node Failure (Min 3 tests)

```
scenario: "Bitcoin node crashes"
```

```yaml
failure: "Bitcoin Core crash"
procedure:
  1. Stop Bitcoin node on 1 validator
  2. Validator detects unavailable
  3. Validator marks self unavailable
  4. Others continue (10-of-14)
  5. Restart Bitcoin node
  6. Validator rejoins when synced
duration: "<5 min detection, ~10 min resync"
success:
  - Detects failure
  - Excludes self
  - Network continues
  - Rejoins automatically
```

Test 5: Stacks Node Failure (Min 3 tests)

Similar to Bitcoin node test.

Test 6: HSM Failure (Min 1 test)

```yaml
scenario: "HSM malfunctions"
failure: "HSM not responding"
procedure:
  1. Simulate HSM malfunction
  2. Validator detects unavailable
  3. Retrieve backup from offsite
  4. Decrypt backup
  5. Import to replacement HSM
  6. Verify integrity
  7. Resume operations
duration: "<2 hours"
success:
  - Failure detected
  - Key recovered
  - Replacement initialized
  - Validator resumes
  - Integrity verified (test sig)
```

Testing Schedule:

```
Month 13, Week 1:
  Day 1-2: Key recovery (3 runs)
  Day 3: Partition (5 runs)
  Day 4: Failover (2 runs)
  Day 5: Review

Month 13, Week 2:
  Day 1: Bitcoin failures (3 runs)
```

```
  Day 2: Stacks failures (3 runs)
  Day 3: HSM failure (1 run)
  Day 4: Combined scenario
  Day 5: Review, document

Month 13, Week 3:
  Day 1-2: Finalize procedures
  Day 3-4: Train team on DR
  Day 5: Sign-off
```

Success Criteria: - [ ] All scenarios tested successfully - [ ] Recovery times within targets - [ ] Procedures documented - [ ] Team trained - [ ] Quarterly drill schedule set

---

11. Deployment Plan <– ADD THIS

Timeline: 9 months (Months 8-16)
Budget: $463,000
Deliverables: Complete production deployment

11.1 Deployment Overview

Timeline: 9 months (Months 8-16)
Budget: $463,000
Start: Month 8 (after Phase 1 complete)
Launch: Month 14 (soft), Month 16 (full)
Buffer: Months 17-18 (stabilization)

Why 9 Months? - +2 months validator recruitment/onboarding - +1 month operational tooling & DR testing - Dedicated soft launch month - More realistic for distributed system

11.2 Pre-Deployment Checklist

Technical Completeness: - [ ] All code complete and merged - [ ] Unit tests: >90% coverage - [ ] Integration tests: 100+ scenarios - [ ] Load tests: 1000 users simulated - [ ] Security audit: Zero critical, all high resolved - [ ] Documentation: User docs, API docs, runbooks

Infrastructure Readiness: - [ ] 15 validators recruited and onboarded - [ ] All validators hardware verified - [ ] Testnet deployed and running 30+ days - [ ] 2+ testnet DKG rehearsals successful - [ ] Monitoring: Grafana, PagerDuty operational - [ ] Backups: Automated and tested

Operational Readiness: - [ ] On-call rotation established - [ ] Incident runbooks complete (10+ scenarios) - [ ] DR procedures tested (6 scenarios) - [ ] Communication channels active (Discord, Telegram) - [ ] Emergency contacts 24/7

Business Readiness: - [ ] Legal structure: Foundation established - [ ] Marketing materials: Website, docs, videos - [ ] Community: 2000+ Discord members - [ ] Miner outreach: 10+ mining ops contacted - [ ] Institutional outreach: 5+ VCs/treasuries

11.3 Month-by-Month Timeline

MONTH 8: Foundation & Recruitment

Week 1-4: - [ ] D2.1: Threshold crypto library integration - [ ] D2.1a: Validator recruitment begins (outreach to 50+ candidates) - [ ] D2.2: Validator software development starts

Milestones: - Threshold crypto library selected - 50+ validator candidates contacted - Validator software architecture complete

Budget: $45K (10%)

MONTH 9: Development & Network Building

Week 1-4: - [ ] D2.2: Validator software continues - [ ] D2.3: DKG implementation - [ ] D2.4: Bridge logic starts - [ ] D2.1a: Technical interviews (25 candidates)

Milestones: - DKG implementation complete - 25 validators interviewed - Bridge logic 50% complete

Budget: $70K cumulative (15%)

MONTH 10: Integration & Testing

Week 1-4: - [ ] D2.2: Validator software complete - [ ] D2.1a: Validator onboarding (15 selected) - [ ] D2.3a: First testnet DKG rehearsal - [ ] D2.4: Bridge logic continues - [ ] D2.4a: State management begins - [ ] D2.4b: Network protocol begins

Milestones: - 15 validators recruited - Validator software deployed to testnet - First DKG rehearsal successful

Budget: $202K cumulative (44%)

MONTH 11: State Management & Protocols

Week 1-4: - [ ] D2.4: Bridge logic complete - [ ] D2.4a: State management complete - [ ] D2.4b: Network protocol complete - [ ] D2.5: Validator network launch begins - [ ] D2.7: Frontend updates begin - [ ] D2.3a: Second testnet DKG rehearsal

Milestones: - Bridge logic production-ready - All 15 validators on testnet - Second DKG rehearsal successful

Budget: $297K cumulative (64%)

MONTH 12: Security & Coordination

Week 1-4: - [ ] D2.5: Validator network fully deployed (testnet) - [ ] D2.3a: Mainnet DKG ceremony (Week 1) - [ ] D2.6: Security audit begins (Weeks 1-4) - [ ] D2.7: Frontend updates continue

Milestones: - Mainnet DKG ceremony - Combined public key generated - Security audit in progress

Budget: $393K cumulative (85%)

MONTH 13: Audit, Tooling & Final Prep

Week 1-4: - [ ] D2.6: Security audit completes (Week 2) - [ ] Address audit findings (Week 3) - [ ] D2.6: Re-audit (Week 4) - [ ] D2.7: Frontend complete - [ ] D2.8: Migration tools begin - [ ] D2.10: Operational tooling deployment - [ ] D2.11: Disaster recovery testing (all scenarios)

Milestones: - Security audit passed - Operational tooling deployed - DR procedures tested

Budget: $451K cumulative (97%)

MONTH 14: Launch Preparation & Migration

Week 1-2: - [ ] Pre-launch validation (full checklist) - [ ] Mainnet contracts deployed - [ ] Validator network: Mainnet mode

Week 3: - [ ] D2.8: Migration tools complete - [ ] First loans migrated from Phase 1

Week 4: - [ ] **SOFT LAUNCH** - [ ] D2.9: Marketing begins - [ ] Whitelist users (<$50K loans) - [ ] Close monitoring (24/7)

Milestones: - Mainnet deployed - First native BTC loan created - Zero critical bugs

Budget: $458K cumulative (99%)

MONTH 15: Soft Launch & Early Adoption

Week 1-4: - [ ] D2.9: First loans (target: 5+) - [ ] 24/7 monitoring - [ ] User onboarding & support - [ ] Migration support (Phase 1 users) - [ ] Feedback incorporation

Milestones: - $500K-$1M volume - 5+ loans created - 99.9% uptime maintained - Positive user feedback

Budget: $461K cumulative (99.5%)

MONTH 16: Full Launch & Growth

Week 1: - [ ] **FULL PUBLIC LAUNCH** - [ ] Remove loan size limits - [ ] Public announcement (PR, Twitter, blog)

Week 1-4: - [ ] D2.9: Volume growth campaign - [ ] Institutional outreach (miners, VCs) - [ ] Conference presentations - [ ] Migration push (incentives)

Week 4: - [ ] **$5M VOLUME MILESTONE**

Milestones: - $5M volume achieved - 15+ borrowers, 30+ lenders - 50%+ market share (new loans) - Validator network profitable

Budget: $463K (100%)

MONTHS 17-18: Buffer Period

Purpose: Stabilize before Phase 3

Activities: - Continuous monitoring (99.9% uptime) - Performance optimization - User feedback incorporation - Validator performance review - Phase 3 planning and design - Phase 3 grant applications

Target: - 2 months trouble-free operation - No critical incidents - Validator uptime: 99.9% - User satisfaction: 4.5+/5 - Ready for Phase 3 (Month 18)

11.4 DKG Ceremony Coordination (D2.3a)

Timeline: Months 10-12
Budget: $20,000

Testnet Rehearsal 1 (Month 10, Week 3)

Preparation: - [ ] All 15 validators notified (2 weeks advance) - [ ] Calendar invites sent (with timezone conversions) - [ ] Ceremony procedures reviewed - [ ] Communication channels tested (Discord voice) - [ ] Coordinator scripts ready

Day-of Ceremony: 1. Pre-ceremony checks (30 min) - All validators online - Software versions match - Network connectivity verified

2. Ceremony execution (10-15 min)
   - Coordinator initiates DKG
   - All validators participate
   - Key shares generated
   - Combined public key exported
3. Post-ceremony verification (15 min)
   - Public key verified
   - Test transaction signed
   - Audit logs reviewed

Success: Rehearsal completed, issues identified

Testnet Rehearsal 2 (Month 11, Week 4)

Same as Rehearsal 1 but: - Address issues from Rehearsal 1 - Inject 3 validator failures (test resilience) - Verify ceremony completes with 12 validators

Success: Rehearsal successful with failures

Mainnet DKG Ceremony (Month 12, Week 1)

Critical Event: Cannot proceed without success

Preparation (Week before): - [ ] All validators confirm availability - [ ] Backup ceremony date scheduled (if needed) - [ ] Emergency procedures reviewed - [ ] Team on standby (all hands)

Ceremony Day: - Start time: 14:00 UTC (accommodates all timezones) - Duration: 30 minutes scheduled, expect <15 min - Coordinator: Project lead - Backup coordinator: Senior dev

Procedure: 1. T-30min: Pre-ceremony briefing (Discord voice) 2. T-15min: Final checks (all validators ready) 3. T-0: Ceremony begins 4. Key generation: 10-15 minutes 5. Post-ceremony verification: 15 minutes 6. Team debrief: 15 minutes

Contingency: - If ceremony fails: Analyze issues, reschedule for next day - If <15 validators: Ceremony cannot proceed - If >3 hours delay: Reschedule to backup date

Post-Ceremony: - [ ] Public key published - [ ] Test transaction signed and confirmed - [ ] Audit logs archived - [ ] Ceremony report published (public)

11.5 Mainnet Deployment Procedure

Timeline: Month 14, Week 2-3 (3 days)

Day 1: Contract Deployment

Morning (4 hours):

```
# 1. Pre-deployment checks
./scripts/pre_deploy_checks.sh
# Verify: Tests pass, balances sufficient, hashes match

# 2. Deploy main protocol contract
clarinet deploy --network mainnet contracts/loan-protocol-v2.clar
# Verify: Contract deployed, address saved

# 3. Deploy NFT contracts
clarinet deploy --network mainnet contracts/borrower-nft.clar
clarinet deploy --network mainnet contracts/lender-nft.clar

# 4. Deploy marketplace
clarinet deploy --network mainnet contracts/marketplace.clar
```

Afternoon (4 hours):

```
# 5. Initialize stablecoin whitelist
clarinet run --network mainnet scripts/init_stablecoins.ts
# Add: USDA, USDC, xUSD

# 6. Configure validator registry
clarinet run --network mainnet scripts/register_validators.ts
# Register: All 15 validators with public keys

# 7. Set admin controls
clarinet run --network mainnet scripts/set_admin.ts
# Configure: 3-of-5 multisig
```

End of Day: - [ ] All contracts deployed - [ ] Stablecoins whitelisted - [ ] Validators registered - [ ] Smoke tests passed

---

Day 2: Infrastructure Deployment

Morning (4 hours):

```
# 1. Deploy backend API
cd backend
docker-compose -f docker-compose.prod.yml up -d
# Services: API, PostgreSQL, Redis
```

```
# 2. Deploy frontend
cd frontend
npm run build
vercel deploy --prod
# Or: netlify deploy --prod

# 3. Configure DNS
# Update: api.protocol.example → API server IP
# Update: app.protocol.example → Vercel/Netlify

# 4. SSL certificates
certbot certonly --dns-cloudflare -d api.protocol.example
```

Afternoon (4 hours):

```
# 5. Deploy monitoring (Grafana)
cd monitoring
docker-compose up -d
# Services: Grafana, Prometheus, Alertmanager

# 6. Deploy ELK stack
cd logging
docker-compose up -d
# Services: Elasticsearch, Logstash, Kibana

# 7. Configure PagerDuty
# Import: alert_rules.yaml
# Test: Send test alert

# 8. Validator health checks
for validator in validators/*; do
  ./scripts/health_check.sh $validator
done
```

End of Day: - [ ] Backend deployed - [ ] Frontend deployed - [ ] Monitoring operational - [ ] Logging aggregated - [ ] All validators healthy

---

Day 3: Validation & Soft Launch

Morning (4 hours):

```
# 1. End-to-end validation
./scripts/e2e_validation.sh
# Tests: Create loan, bid, finalize, repay

# 2. Load testing
k6 run load_tests/mainnet_launch.js
```

```
# Simulate: 100 concurrent users

# 3. Monitor for issues
# Watch: Grafana dashboards, logs, alerts

# 4. Final go/no-go decision
# Team meeting: Review all metrics
```

Afternoon (4 hours):

```
# 5. Soft launch announcement
# Post: Discord, Twitter, blog
# Audience: Whitelist users only

# 6. First loans
# Monitor: First 5 loans created
# Support: Real-time user assistance

# 7. 24/7 monitoring begins
# Rotation: 2 engineers on-call at all times

# 8. Incident response ready
# Verify: All runbooks accessible
```

End of Day: - [ ] Mainnet validated - [ ] Soft launch announced - [ ] First loans created - [ ] 24/7 monitoring active

---

11.6 Rollback Plan

SEV-1: Funds at Risk

Immediate actions: 1. Emergency pause (smart contract) 2. All hands on deck 3. Investigate root cause 4. Develop fix 5. Re-audit fix (expedited) 6. Deploy fix 7. Unpause

Timeline: 1-3 days

SEV-2: Broken Functionality

Actions: 1. Log issue 2. Continue operating (if safe) 3. Develop fix 4. Test in staging 5. Deploy patch

Timeline: 1 week

SEV-3: Minor Issue

Actions: 1. Log issue 2. Queue for next release 3. Deploy in maintenance window

Timeline: 1-4 weeks

11.7 Success Criteria

Before Proceeding to Next Phase: - [ ] Mainnet deployed successfully - [ ] Zero critical bugs first 2 weeks - [ ] 10+ loans created - [ ] 5+ loans repaid - [ ] All 3 stablecoins working - [ ] Validator uptime >99% - [ ] $1M+ volume by Month 16 end

---

12. Monitoring & Operations

12.1 Monitoring Dashboard

Validator Health:

```
Validator Network Status


Online: 14/15 (93%)
Threshold: 10 required
Current Capacity: 140%

Validators:
[1]       99.8% ↑ 2s
[2]       99.9% ↑ 1s
[3]       99.7% ↑ 3s
[4]       99.6% ↑ 5s
[5]       98.2% ↓ 15m
...

Recent Signatures:
• Loan #157: Released (12 sigs, 2m)
• Loan #155: Released (11 sigs, 3m)
• Loan #154: Claimed (13 sigs, 2m)
```

Signing Performance:

```
Threshold Signing Performance
```

```
Last 24 Hours:
• Signatures: 47
• Success Rate: 100%
• Avg Time: 2m 15s
• P95: 3m 45s
• P99: 4m 30s

Current Queue:
• Pending: 2 sessions
• Oldest: 45 seconds
```

Bitcoin/Stacks Sync:

```
Blockchain Synchronization


Bitcoin:
• Height: 825,432  Synced
• Mempool: 2,347 txs
• Avg Fee: 45 sat/vB

Stacks:
• Height: 145,678  Synced
• Anchored to Bitcoin: 825,432

Deposits Pending Confirmation:
• Loan #158: 2/3 confirmations
```

12.2 Alerting Rules

P0 Alerts (Page Immediately):

```
- Validators online < 10 (below threshold)
- Signing failure (unable to generate signature)
- Bitcoin deposit not confirmed after 6 hours
- Security incident detected
- Key share access anomaly
```

P1 Alerts (Response within 1 hour):

```
- Validator down for >15 minutes
```

```
- Signing latency >5 minutes (P95)
- Bitcoin reorg detected (>3 blocks)
- Network partition suspected
- Abnormal signing activity
```

P2 Alerts (Response within 4 hours):

```
- Validator slow response (>99% to 95-99%)
- Low peer connectivity (<8 peers)
- Bitcoin/Stacks node behind (>10 blocks)
- High Bitcoin fees (>100 sat/vB)
```

12.3 Runbook: Common Scenarios

Scenario: Validator Down

```
Detection: Heartbeat missing for >5 minutes

Actions:
1. Check validator server status
   - SSH to server
   - Check systemd status: systemctl status validator
   - Review logs: journalctl -u validator -n 100

2. If process crashed:
   - Identify crash reason from logs
   - Restart: systemctl restart validator
   - Monitor for successful startup

3. If server down:
   - Contact hosting provider
   - Bring server back online
   - Verify blockchain sync
   - Restart validator process

4. If >4 validators down (critical):
   - Page all on-call team
   - Escalate to incident commander
   - Activate backup validators if available
   - Consider emergency pause if <10 online

Expected Resolution Time: 15-30 minutes
```

Scenario: Signing Timeout

```
Detection: Signing session >5 minutes without completion

Actions:
```

```
1. Check signing session status:
   GET /api/v1/signing/session/:id

2. Identify non-responsive validators:
   - Review nonce_commitments and partial_signatures
   - Identify missing validators

3. Contact non-responsive validator operators
   - Request status check
   - Identify issue (network, process, etc.)

4. If sufficient signatures (10+):
   - Coordinator can proceed with finalization
   - Log non-responsive validators

5. If insufficient signatures:
   - Retry signing session
   - Exclude non-responsive validators
   - Use different validator subset

6. Post-Incident:
   - Review why validators were non-responsive
   - Update monitoring if needed
   - Consider validator rotation if persistent

Expected Resolution Time: 5-15 minutes
```

Scenario: Bitcoin Reorg

```
Detection: Bitcoin block height decreased or block hash changed

Actions:
1. Determine reorg depth:
   - Compare current chain to our last known
   - Identify affected blocks

2. Identify affected loans:
   - Query loans with deposits in reorged blocks
   - Check if any confirmations were reversed

3. For each affected loan:
   - If deposit was removed:
     * Update Stacks contract status back to "awaiting-deposit"
     * Notify borrower
     * Monitor for deposit re-confirmation

   - If deposit still exists (different block):
```

```
     * Update confirmation count
     * Continue normal process

4. Monitor for chain stability:
   - Wait for 6+ confirmations before proceeding
   - Verify no further reorgs

5. Document incident:
   - Reorg depth, duration, affected loans
   - Any funds at risk (should be none)
   - Lessons learned

Expected Resolution Time: 1-2 hours
```

12.4 Operational Tooling Architecture (D2.10)

Budget: $15,000
Timeline: Month 13

12.4.1 Overview Production-grade monitoring for 15 distributed validators: - Real-time visibility (Grafana) - Proactive alerting (PagerDuty) - Performance metrics (Prometheus) - Log aggregation (ELK stack) - Automated health checks - Incident response

12.4.2 Grafana Dashboard Suite 10+ Dashboards:

Dashboard 1: Validator Health

```yaml
dashboard: "Validator Health Overview"
panels:
  - Validator Status Cards (15 cards)
    - Green: Online, signing
    - Yellow: Online, slow
    - Red: Offline or failing
  - Uptime Percentage (per validator, 30 days)
  - Last Signature Time (per validator)
  - Resource Usage (CPU, RAM, disk per validator)
  - Alert History (last 24 hours)
refresh: "10s"
```

Dashboard 2: Network Performance

```yaml
dashboard: "Network Performance"
panels:
  - Active Connections (gauge, target: 14 per validator)
  - Network Latency (p50, p95, p99 per peer)
```

```
  - Message Throughput (messages/sec)
  - Gossip Propagation Time (histogram)
  - Packet Loss Rate (percentage)
  - Bandwidth Usage (Mbps per validator)
refresh: "30s"
```

Dashboard 3: Signature Requests

```
dashboard: "Signature Request Monitoring"
panels:
  - Request Queue Length (gauge)
  - Requests per Minute (graph, 24h)
  - Average Signature Time (p50, p95)
  - Partial Signatures Received (counter per validator)
  - Aggregation Success Rate (percentage)
  - Failed Signatures (count, with reasons)
refresh: "5s"
```

Dashboard 4: Cross-Chain State

```
dashboard: "Bitcoin  Stacks State"
panels:
  - Bitcoin Height (current)
  - Stacks Height (current)
  - State Sync Lag (blocks)
  - Reorgs Detected (count, last 7 days)
  - Pending Deposits (count)
  - Pending Withdrawals (count)
  - State Consistency Checks (pass/fail)
refresh: "30s"
```

Dashboard 5-10: Blockchain monitoring, UTXO tracking, loan status, user activity, error rates, system resources

12.4.3 PagerDuty Integration 20+ Alert Types:

P1 Alerts (Critical):

```
- name: "Validator Offline"
  condition: "validator_online == 0 for 5 minutes"
  action: "Page on-call engineer immediately"

- name: "Signature Failure Rate High"
  condition: "signature_failure_rate > 5% for 5 minutes"
  action: "Page on-call + escalate to lead"

- name: "Funds at Risk"
  condition: "unauthorized_transaction_detected"
  action: "Page entire team + emergency procedures"
```

```
- name: "State Inconsistency"
  condition: "state_consistency_check_failed"
  action: "Pause transactions, page on-call"

- name: "Critical Validators Down"
  condition: "failed_validators >= 6"
  action: "Cannot sign, page entire team"
```

P2 Alerts (High):

```
- name: "High Latency"
  condition: "signature_time_p95 > 20s for 10 minutes"
  action: "Notify on-call (no page)"

- name: "Reorg Detected"
  condition: "bitcoin_reorg_detected"
  action: "Notify team, monitor handling"

- name: "Validator Performance Degraded"
  condition: "validator_uptime < 99% for 24 hours"
  action: "Notify validator operator"
```

P3 Alerts (Medium):

```
- name: "Resource Usage High"
  condition: "cpu_usage > 80% for 30 minutes"
  action: "Create ticket, investigate"

- name: "Disk Space Low"
  condition: "disk_free < 20%"
  action: "Notify DevOps"
```

On-Call Rotation: - Primary: 1 week shifts - Secondary: Backup for primary - Coverage: 24/7/365 - Response time: <5 minutes (P1), <30 minutes (P2)

12.4.4 Prometheus Metrics Validator Metrics:

```
validator_online{validator_id="validator-001"} 1
validator_uptime_seconds{validator_id="validator-001"} 2419200
validator_signatures_signed{validator_id="validator-001"} 1247
validator_signatures_failed{validator_id="validator-001"} 0
validator_cpu_usage{validator_id="validator-001"} 0.23
validator_memory_usage_bytes{validator_id="validator-001"} 4294967296
validator_disk_usage_bytes{validator_id="validator-001"} 536870912000
```

Network Metrics:

```
network_active_connections{validator_id="validator-001"} 14
```

```
network_latency_seconds{validator_id="validator-001",peer="validator-002",quantile="0.95"} 0
network_messages_sent{validator_id="validator-001"} 45678
network_messages_received{validator_id="validator-001"} 45123
network_bytes_sent{validator_id="validator-001"} 123456789
network_bytes_received{validator_id="validator-001"} 119876543
```

Signature Metrics:

```
signature_requests_total 1247
signature_requests_active 3
signature_time_seconds{quantile="0.5"} 4.2
signature_time_seconds{quantile="0.95"} 8.7
signature_time_seconds{quantile="0.99"} 12.3
partial_signatures_received{validator_id="validator-001"} 1247
final_signatures_produced 1247
signature_failures_total 0
```

Retention: 30 days detailed, 1 year aggregated

12.4.5 ELK Stack for Logging Elasticsearch:

```yaml
cluster:
  nodes: 3
  storage: 500GB per node
indices:
  - validator-logs-*
  - bitcoin-logs-*
  - stacks-logs-*
  - api-logs-*
retention: 90 days
```

Logstash Pipeline:

```
input {
  # Collect from all validators
  beats {
    port => 5044
  }
}

filter {
  # Parse JSON logs
  json {
    source => "message"
  }

  # Add metadata
  mutate {
```

```
    add_field => {
      "[@metadata][index]" => "validator-logs-%{+YYYY.MM.dd}"
    }
  }
}

output {
  elasticsearch {
    hosts => ["es01:9200", "es02:9200", "es03:9200"]
    index => "%{[@metadata][index]}"
  }
}
```

Kibana Queries: - Search: Full-text across all logs - Filter: By validator, severity, timerange - Visualize: Error rates, patterns - Alert: On log patterns

12.4.6 Automated Health Checks Every 1 Minute:

```
#!/bin/bash
# health_check.sh

# Check validator online
if ! curl -sf http://validator-001:9090/health > /dev/null; then
  alert "Validator offline: validator-001"
fi

# Check signing capability
if ! curl -sf http://validator-001:9090/can-sign > /dev/null; then
  alert "Validator cannot sign: validator-001"
fi

# Check resource usage
cpu=$(curl -s http://validator-001:9090/metrics | grep cpu_usage | awk '{print $2}')
if (( $(echo "$cpu > 0.9" | bc -l) )); then
  alert "High CPU on validator-001: ${cpu}%"
fi
```

Every 5 Minutes: - Network connectivity (ping all peers) - Bitcoin node sync status - Stacks node sync status - Database connectivity - API endpoint health

Every 1 Hour: - Backup verification - Disk space check - Log rotation status - Certificate expiry check

12.4.7 Incident Response Runbooks 10+ Scenarios Documented:

Runbook 1: Validator Offline

```
# Incident: Validator Offline
```

```
## Symptoms
- PagerDuty alert: "Validator Offline"
- Grafana dashboard shows red status
- No heartbeat for >5 minutes

## Impact
- Signing capacity reduced
- If  6 offline, cannot sign

## Investigation
1. Check validator logs (Kibana)
2. SSH to validator server
3. Check process status: `ps aux | grep validator`
4. Check system resources: `top`, `df -h`
5. Check network: `ping 8.8.8.8`

## Resolution
- Process crashed: `systemctl restart validator`
- Out of disk: Clear logs, expand volume
- Network issue: Contact ISP, fail over to backup connection
- Hardware failure: Fail over to backup validator

## Prevention
- Set up resource alerts
- Enable automatic restarts
- Configure backup connections

## Post-Incident
- Create postmortem
- Update monitoring
- Share learnings with team
```

Runbook 2-10: Signature failures, state inconsistency, reorg handling, network partition, DDoS attack, certificate expiry, database corruption, backup failure, security incident

12.5.8 Backup Automation Key Share Backups:

```
#!/bin/bash
# backup_key_shares.sh (runs daily)

# Encrypt key share
gpg --encrypt --recipient admin@protocol.example \
  /var/lib/validator/key_share.dat
```

```
# Upload to multiple locations
aws s3 cp key_share.dat.gpg s3://backups/keys/validator-001/$(date +%Y%m%d)/
rsync -avz key_share.dat.gpg backup-server:/backups/keys/validator-001/
scp key_share.dat.gpg offsite-backup:/encrypted/keys/

# Verify upload
if aws s3 ls s3://backups/keys/validator-001/$(date +%Y%m%d)/key_share.dat.gpg; then
  log "Backup successful"
else
  alert "Backup failed for validator-001"
fi

# Cleanup old backups (>90 days)
find /backups/keys/validator-001/ -mtime +90 -delete
```

Database Backups:

```
# backup_database.sh (runs every 6 hours)

# PostgreSQL backup
pg_dump -h localhost -U validator -d validator_db \
  | gzip > /backups/db/validator_db_$(date +%Y%m%d_%H%M).sql.gz

# Upload to S3
aws s3 cp /backups/db/validator_db_$(date +%Y%m%d_%H%M).sql.gz \
  s3://backups/databases/

# Test restore (weekly)
if [ $(date +%u) -eq 1 ]; then
  gunzip < latest_backup.sql.gz | psql -h test-db -U validator -d test_db
  # Run validation queries
fi
```

Backup Verification: - Daily: Verify backups exist - Weekly: Test restore procedure - Monthly: Full DR drill

12.4.9 Success Criteria (D2.10)

- ☐ All 15 validators reporting to Grafana
- ☐ 10+ dashboards operational
- ☐ PagerDuty integrated (20+ alert types)
- ☐ ELK stack aggregating logs
- ☐ Health checks automated (1 min, 5 min, 1 hour)
- ☐ 10+ runbooks documented and tested
- ☐ Backups automated and verified
- ☐ Team trained on all tools

---

13. Disaster Recovery

13.1 Key Recovery Procedures

Scenario: Validator Key Share Lost

```
Prerequisites for Recovery:
- Validator identity verified (multiple factor authentication)
- Backup key share available (from secure storage)
- Recovery ceremony authorized by protocol governance

Recovery Process:
1. Verify validator identity through multiple channels
2. Retrieve backup key share from secure storage
   (requires 3-of-5 Shamir shares)
3. Decrypt backup using validator's master password
4. Load key share into new HSM
5. Verify key share is correct:
   - Compute public key share
   - Compare with on-chain registry
6. Resume validator operations
7. Test signing with other validators

Timeline: 2-4 hours
Risk Level: Medium (requires multiple authorization steps)
```

Scenario: Multiple Validators Lost (6-10)

```
This is a CRITICAL scenario approaching threshold failure.

Immediate Actions (First Hour):
1. Emergency protocol activation
2. Page all validator operators
3. Assess situation:
   - Which validators are affected?
   - Is this coordinated attack or coincidence?
   - Can validators be recovered quickly?

Short-term Mitigation (Hours 1-6):
1. Activate backup validators if available
2. Expedite recovery of lost validators
3. Pause new loan creation if below threshold
4. Allow existing loans to complete
5. Coordinate emergency validator recruitment

Long-term Recovery (Days 1-7):
1. Add emergency validators (if needed):
   - Community members
```

```
   - Partner organizations
   - Temporary commercial validators
2. Perform new DKG ceremony (if threshold broken)
3. Migrate to new validator set (if necessary)
4. Post-mortem and prevention measures

Timeline: 1-7 days depending on severity
Risk Level: CRITICAL
```

13.2 Contract Upgrade Path

Phase 2 Contracts are Immutable: - Smart contracts cannot be upgraded - If critical bug found, must deploy new version - Existing loans continue on old contract - New loans use new contract

Migration Strategy:

```
1. Deploy new contract version
2. Mark old contract as deprecated
3. Allow both versions to run simultaneously
4. Migrate users gradually:
   - Existing loans complete normally on old contract
   - New loans go to new contract
5. After all old loans complete, old contract becomes inactive
```

Emergency Pause Function:

```
;; Emergency pause (only for new loans)
(define-data-var emergency-paused bool false)

(define-public (set-emergency-pause (paused bool))
  (begin
    (asserts! (is-eq tx-sender (var-get contract-owner)) ERR_UNAUTHORIZED)
    (var-set emergency-paused paused)
    (ok true)
  )
)

;; Check before allowing new loans
(asserts! (not (var-get emergency-paused)) ERR_CONTRACT_PAUSED)
```

13.3 Communication Plan

Incident Severity Levels:

SEV-1: Critical

```
Examples:
- Validators below threshold (<10 online)
```

```
- Funds at risk
- Security breach

Communication:
- Immediate: Discord announcement (within 15 minutes)
- Website banner
- Twitter/X announcement
- Email to all users
- Update every hour until resolved

Stakeholders:
- All users
- Validator operators
- Core team
- Investors/partners
- Media (if funds at risk)
```

SEV-2: Major

```
Examples:
- Signing delays (>10 minutes)
- 3-5 validators down (but above threshold)
- Bitcoin reorg affecting loans

Communication:
- Discord announcement (within 1 hour)
- Status page update
- Email to affected users
- Update every 4 hours

Stakeholders:
- Affected users
- Validator operators
- Core team
```

SEV-3: Minor

```
Examples:
- Single validator down
- Planned maintenance
- Non-critical bugs

Communication:
- Status page update
- Discord post
- Regular status updates

Stakeholders:
```

```
- Core team
- Validator operators (if relevant)
```

Template: SEV-1 Incident:

```
INCIDENT ALERT [SEV-1]

Title: [Brief description]
Time Detected: [Timestamp UTC]
Status: INVESTIGATING / IDENTIFIED / MONITORING / RESOLVED

Impact:
- [What is affected]
- [How many users/loans affected]
- [Are funds at risk? YES/NO]

Actions Taken:
- [Step 1]
- [Step 2]

Next Steps:
- [What we're doing now]
- [ETA for next update]

User Action Required:
- [What users should do, if anything]

We apologize for any inconvenience. Updates will be posted every hour
until resolution.

[Link to status page for updates]
```

13.4 Disaster Recovery Testing (D2.11)

Budget: $12,000
Timeline: Month 13

13.4.1 Overview Comprehensive testing of all disaster scenarios to ensure validator network resilience and recovery capabilities.

6 Test Scenarios: 1. Key Recovery (3 validators lose keys) 2. Network Partition (10-5 split) 3. Coordinator Failover 4. Bitcoin Node Failures 5. Stacks Node Failures 6. HSM Failures

Each scenario tested multiple times with documented procedures.

13.4.2 Detailed Test Procedures (Test procedures already documented in Section 10.X)

13.4.3 Emergency Procedures Procedure 1: Emergency Pause

```
# When to use: Funds at risk, critical vulnerability discovered

# 1. Activate pause
clarinet call loan-protocol-v2 emergency-pause

# 2. Notify all validators
./scripts/notify_validators.sh "EMERGENCY PAUSE ACTIVATED"

# 3. Notify users
# Post to: Discord, Twitter, website banner

# 4. Investigate issue
# Gather: Logs, metrics, reproduction steps

# 5. Develop fix
# Test in staging environment

# 6. Security review
# Re-audit if critical

# 7. Deploy fix
# Follow deployment procedure

# 8. Unpause
clarinet call loan-protocol-v2 emergency-unpause

# 9. Monitor closely
# 24-48 hours elevated monitoring
```

Procedure 2: Validator Replacement

```
# When: Validator permanently unavailable or compromised

1. Identify replacement validator (from waitlist)
2. Onboard replacement (standard procedure)
3. Schedule re-DKG ceremony (all 15 validators)
4. Execute DKG ceremony with replacement
5. Generate new combined public key
6. Migrate loans to new multisig address
7. Update validator registry
8. Verify new validator signing
9. Decommission old validator
```

```
10. Update documentation
```

Procedure 3: Mass Failure Event

```
# When: >6 validators fail simultaneously (cannot sign)

1. IMMEDIATE: Emergency pause protocol
2. Assess: Identify failure cause
   - Geographic event (disaster)?
   - Systemic issue (software bug)?
   - Attack (DDoS, coordinated)?
3. Triage:
   - If  10 validators recoverable: Recover them
   - If <10 recoverable: Initiate replacement procedures
4. Recovery:
   - Follow key recovery procedures for each
   - Stagger recovery (don't overload network)
5. Resume:
   - Wait for 10+ validators healthy
   - Run test signatures
   - Unpause protocol
6. Post-mortem:
   - Root cause analysis
   - Update procedures
   - Improve resilience
```

13.4.4 Recovery Time Objectives

Scenario	RTO	RPO	Priority
Single validator failure	<1 hour	0	Medium
3 validator failures	<2 hours	0	High
5 validator failures	<4 hours	0	Critical
Network partition	<2 min	0	Critical
Coordinator failover	<30 sec	0	Critical
Key loss (single)	<2 hours	0	High
Key loss (multiple)	<4 hours	0	Critical
Database corruption	<5 min (checkpoint)	1 hour	Medium
Database corruption	<2 hours (rebuild)	0	Medium

RTO: Recovery Time Objective (how long to restore)
RPO: Recovery Point Objective (how much data loss acceptable)

13.4.5 Quarterly Drill Schedule Quarter 1 (Months 14-16): - Month 14: Key recovery drill - Month 15: Network partition drill - Month 16: Coordinator failover drill

Quarter 2 (Months 17-19): - Month 17: Combined scenario (partition + key loss) - Month 18: HSM failure drill - Month 19: Mass failure simulation

Quarter 3 (Months 20-22): - Month 20: Bitcoin reorg + state recovery - Month 21: Validator replacement procedure - Month 22: Emergency pause/unpause

Ongoing: - Every quarter: Full disaster recovery review - Every 6 months: Update all runbooks - Annually: Comprehensive DR audit

13.4.6 Success Criteria (D2.11)

- ☐ All 6 scenarios tested successfully
- ☐ Recovery times within RTOs
- ☐ All procedures documented
- ☐ Team trained on all procedures
- ☐ Quarterly drill schedule established
- ☐ Emergency contacts 24/7
- ☐ Runbooks accessible (printed + digital)

Appendix A: FROST Algorithm Details

Mathematical Specification

Setup Parameters: - Group: secp256k1 (Bitcoin's elliptic curve) - Generator: G - Order: n (large prime) - Participants: n = 15 - Threshold: t = 10

Key Generation (DKG):

```
Each participant i (i = 1..15):
1. Choose random secret: s_i  Z
2. Create polynomial of degree t-1:
   f_i(x) = s_i + a_{i,1}x + a_{i,2}x² + ... + a_{i,t-1}x^(t-1)
3. Compute commitments: C_{i,j} = a_{i,j} * G for j = 0..t-1
4. Broadcast commitments to all participants
5. Send share to each participant j: f_i(j) (encrypted)

Each participant j verifies received shares:
For each share f_i(j) from participant i:
  Check: f_i(j) * G =  (C_{i,k} * j^k) for k = 0..t-1

Final key share for participant j:
  s_j =  f_i(j) for i = 1..15

Public key:
  P =  C_{i,0} =  (s_i * G) for i = 1..15
```

Signing:

```
Message: m
Participants: S   {1..15} where |S| >= 10

Nonce Generation (each participant i   S):
1. Choose random: (d_i, e_i)
2. Compute: (D_i, E_i) = (d_i * G, e_i * G)
3. Broadcast: (D_i, E_i)

Signature Generation:
1. Compute binding values:  _i = H(i, m, {D_j, E_j})
2. Compute group commitment: R =  (D_i +  _i * E_i) for i   S
3. Compute challenge: c = H(R || P || m)
4. Compute Lagrange coefficient for each i   S:
    _i =  (j / (j - i)) for j   S, j   i
5. Each participant i creates partial signature:
   z_i = d_i + (e_i *  _i) + ( _i * s_i * c)
6. Combine: z =   z_i for i   S
7. Final signature: (R, z)

Verification:
  z * G = R + c * P
```

Appendix B: Validator Hardware Specifications

Minimum Requirements:

```
CPU: 8 cores (3.0 GHz+)
RAM: 32 GB
Storage: 2 TB NVMe SSD
Network: 100 Mbps symmetric
Uptime: 99.5%
Operating System: Ubuntu 22.04 LTS

Estimated Cost: $200-300/month (cloud)
                $2,000-3,000 (self-hosted hardware)
```

Recommended Setup:

```
CPU: 16 cores (3.5 GHz+)
RAM: 64 GB
Storage: 4 TB NVMe SSD (RAID 1)
Network: 1 Gbps symmetric, redundant connections
UPS: Battery backup
HSM: YubiHSM 2 or equivalent
Uptime: 99.9%
```

```
Estimated Cost: $400-600/month (cloud)
                $5,000-8,000 (self-hosted hardware)
```

Geographic Distribution: - No more than 3 validators in same data center - No more than 5 validators in same country - Minimum 3 continents represented - Diverse cloud providers (AWS, GCP, Azure, OVH, Hetzner, etc.)

Appendix C: Glossary

DKG (Distributed Key Generation): Protocol to generate threshold keys without trusted dealer

FROST: Flexible Round-Optimized Schnorr Threshold signatures

HSM: Hardware Security Module for secure key storage

Partial Signature: Signature fragment created by single validator using key share

Schnorr Signature: Bitcoin's native signature scheme (post-Taproot)

Taproot: Bitcoin upgrade enabling efficient threshold signatures

Threshold Signature: Single signature created by M-of-N parties

Validator: Node operator who holds key share and participates in signing

VSS (Verifiable Secret Sharing): Technique ensuring shares distributed correctly

---

End of Technical Specification Document - Phase 2

Document Version: 1.0
Last Updated: January 12, 2026

This technical specification provides complete implementation details for Phase 2 with Bitcoin-native custody using threshold signatures.