HYYPERLINK

Key Discovery Mining: A New Paradigm for Digital Currency

D. A. Good
Hyyperlink Foundation
contact@hyyperlink.com

November 9, 2024

https://hyyperlink.com

Contents

HYYPERLINK

Key Discovery Mining: A New Paradigm for Digital Currency

D. A. Good

Hyyperlink Foundation

contact@hyyperlink.com

D. A. Good
Hyyperlink Foundation
contact@hyyperlink.com

November 9, 2024

Hyyperlink introduces a novel approach to digital currency that fundamentally reimagines how value is created and distributed in decentralized networks. By combining directed acyclic graph (DAG) architecture with an innovative token generation mechanism called Key Discovery Mining, the system achieves true parallelism without mining pools or competition. Starting from zero tokens, Hyyperlink creates a self-regulating economy through the natural interplay of token generation and fee burning, while its reputation-based peer-2-peer propagation system ensures network health and scalability.

1 Overview

Hyyperlink is a peer-to-peer digital currency system built on a directed acyclic graph (DAG) architecture. The system uses cryptographic key discovery for token generation and implements a reputation-based transaction propagation mechanism to prevent spam and maintain network health.

2 Network Architecture

2.1 P2P Network

The network is built on libp2p[1] with both TCP and WebSocket transport support. Node discovery uses a Kademlia distributed hash table (DHT)[2] - a decentralized system that provides a lookup service similar to a hash table, where the responsibility for maintaining key mappings is distributed among nodes in a way that minimizes disruption from node changes.

Each node maintains:

- A target of 8 peer connections
- Persistent connections to validator nodes
- Regular health checks on connections

Node initialization process:

1. Create or load P2P keypair from config directory
2. Initialize libp2p host with TCP and WebSocket transports
3. Set up protocol handlers for all supported protocols
4. Bootstrap DHT with validator nodes
5. Begin periodic peer discovery

2.2 Network Protocols

The system implements five core protocols:

FullSyncProtocol

- Handles initial graph synchronization
- Uses batched transaction transfer
- Implements retry mechanism with exponential backoff
- Maximum 3 retry attempts per sync operation

StateHashProtocol

- Quick state verification between peers
- Uses SHA3-256 of all transaction hashes
- Triggers full sync on mismatch

TransactionProtocol

- Handles new transaction propagation
- Implements reputation-based forwarding
- Maintains connection pools for efficient broadcasting

MissingTransactionsProtocol

- Retrieves specific transactions by hash
- Uses batched requests/responses
- Handles partial fulfillment of requests

TransactionOrderProtocol

- Manages transaction ordering proposals
- Validator-specific protocol
- Ensures DAG convergence

[1] `https://libp2p.io/`

[2] `https://en.wikipedia.org/wiki/Distributed_hash_table`

2.3 Peer Selection and Reputation

The system implements a peer selection mechanism based on reputation scores. Each peer's reputation is tracked on a scale from 0.0 to 1.0, with several key thresholds:

- Initial peer reputation: 0.5
- Minimum threshold for transaction relay: 0.3
- Maximum peer reputation: 1.0
- Minimum peer reputation: 0.0

2.3.1 Reputation Scoring

Peer reputation is dynamically adjusted based on behavior:

- **Positive Actions** (+0.05):
  - Successfully propagating valid transactions
  - Maintaining stable connections
  - Providing valid sync responses
- **Negative Actions** (-0.1):
  - Propagating invalid transactions
  - Providing invalid sync data
  - Connection instability

The reputation system is implemented as:

```
func (n *Node) GetPeerReputation(peerID peer.ID) float64 {
    reputation, exists := n.reputations[peerID.String()]
    if !exists {
        return InitialPeerReputation // 0.5
    }
    return reputation
}

func (n *Node) UpdatePeerReputation(peerID peer.ID, change float64) {
    current, exists := n.reputations[peerID.String()]
    if !exists {
        current = InitialPeerReputation
    }

    current += change

    if current > MaxPeerReputation {
        current = MaxPeerReputation
    } else if current < MinPeerReputation {
        current = MinPeerReputation
    }

    n.reputations[peerID.String()] = current
}
```

2.3.2 Transaction Propagation Rules

The system implements different propagation rules based on transaction source and peer reputation:

1. **Priority Transactions**:
   - Mining rewards (from TokenGrantAddress)
   - Transactions from wealthy accounts (balance ¿= PovertyLine)
   - Transactions with valid mining proofs
   - These are propagated to all peers with reputation ¿= 0.3
2. **Standard Transactions**:
   - Must come from wallets with sufficient reputation
   - Only propagated to peers with reputation ¿= 0.3
   - Subject to additional validation checks

The propagation logic is implemented as:

```
func (n *Node) BroadcastTransaction(tx *Transaction) {
    if tx.From == TokenGrantAddress ||
       tx.MiningProof != nil ||
       n.graph.State[tx.From] >= PovertyLine {
        // Priority transaction - broadcast to all qualified peers
        for _, peerID := range n.host.Network().Peers() {
            if n.GetPeerReputation(peerID) >= MinimumPeerReputationThreshold {
                go n.SendTransactionToPeer(tx, peerID)
            }
        }
    } else if n.GetWalletReputation(tx.From) >= MinimumReputationThreshold {
        // Standard transaction - broadcast only if sender has sufficient reputation
        for _, peerID := range n.host.Network().Peers() {
            if n.GetPeerReputation(peerID) >= MinimumPeerReputationThreshold {
                go n.SendTransactionToPeer(tx, peerID)
            }
        }
    }
}
```

2.3.3 Connection Management

The system maintains a target of 8 peer connections, with special handling for validator nodes:

- Persistent connections to validator nodes
- Regular health checks every 30 seconds
- Automatic reconnection attempts on failure
- Connection pooling for efficient message distribution

This reputation-based peer selection system provides several benefits:

- Spam prevention through reputation requirements
- Natural isolation of misbehaving peers
- Efficient resource allocation to reliable peers
- Automatic network self-organization
- Resilience against network attacks

The combination of reputation tracking, dynamic adjustment, and differentiated propagation rules creates a robust peer-to-peer network that can efficiently distribute transactions while maintaining security against various forms of abuse.

3 Consensus Mechanism

3.1 DAG Structure

The system uses a DAG where each transaction must reference 1-4 previous transactions. The reference selection process follows these steps:

3.1.1 Candidate Selection

- Scans recent transactions within 12-hour window
- Prioritizes newer transactions
- Excludes self-references

3.1.2 Reference Selection Algorithm

Algorithm 1 Reference Selection

```
$totalWeight \leftarrow 0$
for $i \leftarrow 0$ to $len(candidateTxs) - 1$ do
    $weight \leftarrow len(candidateTxs) - i$
    $totalWeight \leftarrow totalWeight + weight$
end for
$randomWeight \leftarrow random(0, totalWeight)$
```

3.1.3 Reference Validation

- Ensures minimum reference count
- Verifies reference existence
- Checks reference age
- Prevents circular references

3.2 Transaction Processing

Transactions are processed in parallel with the following rules:

3.2.1 Reference Resolution

- All referenced transactions must exist
- References must be within time window
- First transaction in empty graph exempted

3.2.2 State Application

- Atomic balance updates
- Double-spend prevention
- Fee processing
- Mining reward distribution

4 Token Generation Through Key Discovery

The Hyyperlink token generation system implements a novel approach that is unique among cryptocurrency systems. Instead of competing to process transactions, participants generate Ed25519 key pairs and encode their public keys, searching for specific properties in the resulting public key hashes.

This approach serves multiple purposes:

1. Provides a deterministic and verifiable way to generate new tokens
2. Creates controlled token issuance through tunable difficulty requirements
3. Enables fully independent mining without coordination or competition

Unlike traditional mining systems, key discovery allows parallel token generation without requiring global consensus or competition between miners. Each discovered key meeting the difficulty requirement represents a provably rare artifact that can be independently verified by any participant.

4.1 Key Discovery Verification

The verification process ensures three critical aspects:

1. The winning key meets the difficulty requirement
2. The discoverer can sign with the winning key
3. The miner has properly claimed the reward

This creates an unforgeable chain of custody from discovery to reward:

```
proofMessage := fmt.Sprintf("%s:%s", proof.
      WinningAddress, proof.MinerAddress)

// Verify winning key signature
winningPubKey, err := AddressToPublicKey(proof.
      WinningAddress)
if !ed25519.Verify(winningPubKey, []byte(proofMessage)
      , proof.WinningSignature) {
    return false
}

// Verify miner signature
minerPubKey, err := AddressToPublicKey(proof.
      MinerAddress)
if !ed25519.Verify(minerPubKey, []byte(proofMessage),
      proof.MinerSignature) {
    return false
}

// Verify difficulty requirement
isWinner, difficulty := IsWinningKey(proof.
      WinningAddress, MinMinerDifficulty)
if !isWinner {
    return false
}
```

4.2 Mining Process Details

The mining process follows a simple but effective sequence:

1. Generate a new Ed25519 keypair
2. Take the public key and encode it using Base58:

$$address = base58(ed25519.publicKey) \tag{1}$$

3. Calculate SHA3-256 hash of the Base58-encoded public key:

$$hash = sha3.256(address) \tag{2}$$

4. Count the leading zeros in the resulting hash
5. If the number of leading zeros meets or exceeds the minimum difficulty, a reward is issued proportional to the rarity of the hash

This process creates a provably fair system where:

- The difficulty of finding a winning key is predictable
- The rarity of the key can be instantly verified by any participant
- Rewards scale with the difficulty of the discovered key
- No coordination between miners is required

4.3 Mining Reward Calculation

The mining reward follows a linear scaling formula:

$$reward = R_{base} \cdot (d - d_{min} + 1) \tag{3}$$

Where:

- R_{base} is 1.00000000 tokens
- d_{min} is 5 leading zeros
- d is the number of leading zeros in the hash

This creates an incentive structure where higher difficulties receive proportionally larger rewards:

- 5 zeros = 1.00000000 tokens
- 6 zeros = 2.00000000 tokens
- 7 zeros = 3.00000000 tokens

4.4 Mining Process Implementation

The mining process operates as a multi-threaded system that efficiently utilizes available CPU cores:

1. Generates Ed25519 key pairs
2. Checks if they meet the difficulty requirement
3. Submits proofs when winning keys are found
4. Maintains statistics on keys checked per second

```go
func MiningWorker(node *Node) {
    for {
        // Generate new Ed25519 keypair
        pubKey, privKey, err := ed25519.GenerateKey(rand.Reader)
        if err != nil {
            continue
        }

        // Encode public key
        pubKeyBase58 := base58.Encode(pubKey)

        // Check if key meets difficulty requirement
        hash := sha3.Sum256([]byte(pubKeyBase58))
        difficulty := countLeadingZeros(hash)

        if difficulty >= MinMinerDifficulty {
            // Create and submit mining proof
            proof := &MiningProof{
                WinningAddress:    pubKeyBase58,
                MinerAddress:      node.wallet.Address(),
                WinningSignature:  nil,
                MinerSignature:    nil,
            }

            // Sign proof with winning key
            message := fmt.Sprintf("%s:%s", proof.WinningAddress, proof.MinerAddress)
            proof.WinningSignature = ed25519.Sign(privKey, []byte(message))

            // Sign with miner wallet
            proof.MinerSignature = node.wallet.Sign([]byte(message))

            // Submit proof
            node.SubmitMiningProof(proof)
        }
    }
}
```

4.5 Mining Reward Issuance

When a miner submits a proof, the validator processes it through several stages:

```go
// First, verify the mining proof
proof := tx.MiningProof
proofMessage := fmt.Sprintf("%s:%s", proof.WinningAddress, proof.MinerAddress)

// Verify signatures and difficulty
winningPubKey, _ := AddressToPublicKey(proof.WinningAddress)
if !ed25519.Verify(winningPubKey, []byte(proofMessage), proof.WinningSignature) {
    return false
}

minerPubKey, _ := AddressToPublicKey(proof.MinerAddress)
if !ed25519.Verify(minerPubKey, []byte(proofMessage), proof.MinerSignature) {
    return false
}

isWinner, difficulty := IsWinningKey(proof.WinningAddress, MinMinerDifficulty)
if !isWinner {
    return false
}

// Calculate reward based on difficulty
reward := CalculateMinerReward(difficulty)

// Create and process the token issuance transaction
tx := Transaction{
    From:      TokenGrantAddress,
    Timestamp: time.Now().UnixNano(),
    Changes: []TransactionChange{
        {
            To:     minerAddress,
            Amount: reward - BaseFee,  // Miner pays transaction fee
        },
    },
    Fee:         BaseFee,
    MiningProof: proof,
}

// Apply the state change
if err := g.ApplyTransaction(tx); err != nil {
    return fmt.Errorf("failed to apply mining transaction: %v", err)
}
```

The mining reward issuance process implements multiple security checks before creating new tokens:

1. **Proof Verification**: The system first verifies that both the winning key and miner's key have properly signed the proof message, creating a cryptographic chain of custody.
2. **Difficulty Check**: The winning address is verified to meet the minimum difficulty requirement by counting leading zeros in its hash.
3. **Reward Calculation**: The reward amount is calculated based on the achieved difficulty level, with higher difficulties earning proportionally larger rewards.
4. **Token Creation**: A special transaction is created from the TokenGrantAddress, which has unique privileges to issue new tokens.
5. **Fee Handling**: The transaction includes the standard network fee, which is deducted from the mining reward, maintaining consistency with regular transaction rules.

This process ensures that token creation is:

- Cryptographically secure
- Independently verifiable
- Automatically scaled by difficulty
- Properly recorded in the network state

This process ensures that:

- The mining proof is valid and meets difficulty requirements
- The reward is calculated based on the achieved difficulty
- The token issuance is atomic with proof verification
- The miner pays the standard transaction fee from their reward
- The state change is properly recorded in the DAG

5 Transaction Processing

5.1 Transaction Creation and Validation

Each transaction must reference previous transactions, creating a web of confirmations. The system enforces several rules:

5.1.1 Reference Requirements

- Minimum of 1 reference
- Maximum of 4 references
- References must be within a 12-hour window
- First transaction in network exempted from references

5.1.2 Balance and Fee Rules

- All transactions must pay a base fee (10000 units)
- Total transaction amount must not exceed sender's balance
- Fees are burned (sent to BURN address)

The transaction structure is defined as:

```
type Transaction struct {
    Timestamp   int64
    Hash        string
    From        string
    Signature   string
    Changes     []TransactionChange
    References  []string
    Fee         uint64
    MiningProof *MiningProof
}
```

5.2 Transaction Propagation

The propagation system implements a reputation-based flooding protocol that balances network efficiency with spam prevention. The system uses multiple mechanisms to ensure reliable and secure transaction propagation:

5.2.1 Reputation Scoring

The system maintains two types of reputation scores:

1. **Wallet Reputation** (0.0 - 1.0):
    - Wealthy accounts (above poverty line) automatically receive maximum reputation
    - New wallets start with a base reputation of 1.0
    - Successful transactions increase reputation by 0.1
    - Invalid transactions decrease reputation by 0.2
    - Reputation is weighted by the wallet's balance relative to the poverty line
2. **Peer Reputation** (0.0 - 1.0):
    - New peers start with reputation of 0.5
    - Propagating valid transactions increases reputation by 0.05
    - Propagating invalid transactions decreases reputation by 0.1
    - Minimum threshold of 0.3 required for transaction relay

5.2.2 Propagation Rules

Transactions are propagated differently based on their source:

1. **Priority Transactions**:
    - Mining reward transactions (from TokenGrantAddress)
    - Transactions from accounts above the poverty line
    - These are always propagated to all peers above minimum reputation
2. **Regular Transactions**:
    - Must come from wallets with reputation above 0.5
    - Are only propagated to peers with reputation above 0.3
    - First transaction from new wallets must exceed minimum amount threshold

5.2.3 Network Optimization

The system implements several optimizations for efficient propagation:

- Stream reuse for multiple transactions to the same peer
- Transaction deduplication using a broadcast cache
- Automatic stream health checks and recovery
- Batched transaction processing during sync

5.2.4 Gossip Protocol

A background gossip protocol runs every 10 seconds to ensure transaction propagation reliability:

- Checks for non-broadcasted transactions in the local graph
- Re-attempts propagation of previously failed broadcasts
- Helps network recovery after temporary partitions
- Maintains transaction availability across the network

This multi-layered approach creates a robust propagation system that:

- Prevents spam through reputation requirements
- Prioritizes transactions from proven participants
- Maintains network efficiency through peer scoring
- Ensures reliable transaction propagation
- Recovers automatically from network issues

5.3 Cryptographic Verification

Transactions must meet the following cryptographic requirements:

- Transactions must be signed by sender
- Hash must be correctly computed
- Mining proofs must be valid (for mining transactions)

```
// Hash verification
if tx.Hash != tx.ComputeHash() {
    return fmt.Errorf("invalid transaction hash")
}

// Balance verification
totalAmount := tx.Fee
for _, change := range tx.Changes {
    totalAmount += change.Amount
}
if g.State[tx.From] < totalAmount {
    return fmt.Errorf("insufficient balance")
}

// Signature verification
senderPubKey, err := AddressToPublicKey(tx.From)
if !tx.Verify(senderPubKey) {
    return fmt.Errorf("invalid signature")
}
```

5.4 Stream Management

The system implements stream management:

- Reuses existing healthy streams
- Implements timeout handling
- Provides acknowledgment system

```
// Check existing streams
for _, conn := range n.host.Network().ConnsToPeer(peerID) {
    for _, stream := range conn.GetStreams() {
        if stream.Protocol() == protocol.ID(TransactionProtocol) {
            // Reuse existing stream if healthy
            if err := stream.SetWriteDeadline(time.Now().Add(time.Second)); err == nil {
                if _, err := stream.Write([]byte{0}); err == nil {
                    stream.SetWriteDeadline(time.Time{})
                    existingStream = stream
                    break
                }
            }
            stream.Reset()
        }
    }
}
```

5.5 State Updates

The Hyyperlink system implements atomic state updates to ensure consistency and prevent race conditions when modifying account balances. The state update mechanism is a critical component that:

- Ensures atomic (all-or-nothing) balance updates
- Prevents negative balances
- Handles both regular transactions and mining rewards
- Maintains consistency during concurrent operations

5.5.1 Update Process

The state update process follows several key steps:

1. **Balance Change Collection**: First, all balance changes from the transaction are collected and categorized:
   - Positive changes (credits to receiving accounts)
   - Negative changes (debits from sending accounts)
   - Fee deductions (sent to burn address)
2. **Special Case Handling**: Mining reward transactions from the TokenGrantAddress are processed differently:
   - No balance verification needed (allowed to create new tokens)
   - Direct credit to recipient addresses
   - Fee still collected and burned
3. **Balance Verification**: For regular transactions:
   - Total outgoing amount (including fee) is calculated
   - Sender's balance is verified to be sufficient
   - Prevents any possibility of overdraft
4. **Atomic Application**: Changes are applied atomically:
   - All balance updates happen together
   - If any part fails, entire transaction is rolled back
   - Maintains system consistency

The implementation ensures safe concurrent operation:

```
func (g *Graph) ApplyTransaction(tx Transaction) error {
    balanceChanges := make(map[string]uint64)
    negativeChanges := make(map[string]uint64)

    // Process changes
    if tx.From == TokenGrantAddress {
        // Handle mining rewards
        for _, change := range tx.Changes {
            balanceChanges[change.To] = SafeAdd(balanceChanges[change.To], change.Amount)
        }
    } else {
        // Handle regular transactions
        totalDeduction := tx.Fee
        for _, change := range tx.Changes {
            totalDeduction = SafeAdd(totalDeduction, change.Amount)
            balanceChanges[change.To] = SafeAdd(balanceChanges[change.To], change.Amount)
        }
        negativeChanges[tx.From] = totalDeduction
    }

    // Apply changes atomically
    for address, addition := range balanceChanges {
        currentBalance := g.State[address]
        subtraction := negativeChanges[address]

        if subtraction > 0 {
            if subtraction > currentBalance {
                return fmt.Errorf("negative balance would occur")
            }
            currentBalance -= subtraction
        }

        g.State[address] = SafeAdd(currentBalance, addition)
    }

    return nil
}
```

5.5.2 Safety Mechanisms

The system implements several safety mechanisms to maintain integrity:

1. **Overflow Protection**:
    - All arithmetic operations are checked for overflow
    - Uses safe addition function that panics on overflow
    - Prevents balance corruption from integer overflow
2. **Validation Checks**:
    - Pre-validation of transaction amounts
    - Balance sufficiency verification
    - Fee verification
3. **Error Handling**:
    - Clear error messages for debugging
    - Transaction rollback on any error
    - Logging of all state changes

This comprehensive state management system ensures that the network maintains consistent and accurate account balances while preventing any potential exploitation through race conditions or arithmetic errors. The atomic nature of updates, combined with thorough validation and safety checks, provides a robust foundation for the network's financial operations.

5.6 Reputation System Implementation

The reputation system uses two distinct but interconnected reputation tracking mechanisms: wallet reputation and peer reputation. Each serves a different purpose in maintaining network health.

5.6.1 Wallet Reputation Calculation

The wallet reputation system implements a balance-weighted approach:

```
func (n *Node) GetWalletReputation(address string) float64 {
    balance := n.graph.State[address]
    if balance >= PovertyLine {
        return MaxReputation
    }

    reputation, exists := n.reputations[address]
    if !exists {
        balanceWeight := float64(balance) / float64(PovertyLine)
        return InitialReputation * (1 + balanceWeight)
    }
    return reputation
}
```

This implementation provides several key features:

1. **Wealth-Based Trust**:
    - Accounts with balances above the poverty line (10 HYY) automatically receive maximum reputation
    - This reflects the assumption that accounts with significant stakes are less likely to spam
    - Provides immediate trust for well-funded accounts
2. **Balance Weighting**:
    - For accounts below the poverty line, reputation is weighted by their balance
    - The weighting factor is calculated as: $balanceWeight = balance/PovertyLine$
    - This creates a smooth progression of trust as accounts accumulate funds
3. **New Account Handling**:
    - New accounts start with a base reputation of 1.0
    - This base value is then modified by their balance weight
    - Allows new accounts to participate while maintaining spam protection

5.6.2 Peer Reputation Management

The peer reputation system focuses on network behavior:

```
func (n *Node) UpdatePeerReputation(peerID peer.ID, change float64) {
    current, exists := n.reputations[peerID.String()]
    if !exists {
        current = InitialPeerReputation
    }

    current += change

    if current > MaxPeerReputation {
        current = MaxPeerReputation
    } else if current < MinPeerReputation {
        current = MinPeerReputation
    }

    n.reputations[peerID.String()] = current
}
```

This implementation provides:

1. **Bounded Reputation**:
    - Reputation is capped between 0.0 and 1.0

- Prevents reputation inflation or underflow
- Ensures consistent behavior across the network

2. **Gradual Trust Building**:
   - New peers start at 0.5 reputation
   - Small positive increments (+0.05) for good behavior
   - Larger negative increments (-0.1) for bad behavior
   - Creates bias toward conservative trust
3. **Persistent Memory**:
   - Reputation persists across connection sessions
   - Allows network to remember reliable peers
   - Helps maintain stable peer relationships

5.6.3 Reputation Impact on Network Operations

The reputation scores directly influence network behavior:

1. **Transaction Propagation**:
   - Transactions from high-reputation wallets are prioritized
   - Only peers with reputation ¿= 0.3 receive transaction broadcasts
   - Creates natural spam filtering at network level
2. **Peer Selection**:
   - Higher reputation peers are preferred for connection maintenance
   - Low reputation peers may be disconnected during network pruning
   - Helps maintain optimal network topology
3. **Resource Allocation**:
   - Connection slots are preferentially allocated to high-reputation peers
   - Sync requests from high-reputation peers are prioritized
   - Optimizes network resource utilization

This dual reputation system creates a self-regulating network where both account behavior and peer performance contribute to overall network health. The system is designed to be:

- **Self-balancing**: Poor behavior is naturally penalized while good behavior is rewarded
- **Attack-resistant**: Multiple reputation factors make network abuse expensive
- **Performance-oriented**: Resources are automatically allocated to reliable participants
- **Recovery-capable**: Reputation can be rebuilt through consistent good behavior

6 State Management

6.1 Graph State

The graph state represents the current network consensus, tracking:

1. Transaction History:
   - Complete DAG of all transactions
   - Reference relationships
   - Temporal ordering
2. Account Balances:
   - Current balance for all addresses
   - Atomic updates during transaction processing
   - Double-spend prevention
3. Validator Set:
   - Current validator nodes
   - Validator status tracking
   - Update history

6.2 Database Management

The persistence layer uses SQLite as its storage backend, implementing several critical optimizations and safety measures to ensure reliable operation at scale.

6.2.1 Performance Optimizations

The system implements several performance-focused features:

- **Batch Processing**:
  - Transactions are collected into batches of 100
  - Reduces disk I/O overhead
  - Improves throughput under high load
- **Prepared Statements**:
  - SQL statements are pre-compiled
  - Reduces parsing overhead
  - Prevents SQL injection vulnerabilities
- **Transaction Pooling**:
  - Reuses database connections
  - Minimizes connection overhead
  - Manages concurrent access efficiently

6.2.2 Reliability Mechanisms

The database layer implements multiple reliability features to ensure data integrity:

1. **Atomic Commits**:
   - Database transactions are all-or-nothing
   - Batch operations are atomic
   - Prevents partial updates during failures
2. **Rollback Protection**:
   - Automatic rollback on errors

  - Transaction boundary management
  - State consistency preservation

3. **Corruption Prevention**:
   - Safe file handling practices
   - Directory traversal protection
   - Proper permission management

The implementation ensures safe database operations:

```
func ProcessBatch() {
    batchMutex.Lock()
    batch := transactionBatch
    transactionBatch = nil
    batchMutex.Unlock()

    db := GetDB()
    dbTx, err := db.Beginx()
    if err != nil {
        LogError("Failed to begin database transaction: %v", err)
        return
    }

    stmt, err := dbTx.Prepare(‘INSERT OR REPLACE INTO transactions (hash, data) VALUES (?, ?)‘)
    if err != nil {
        LogError("Failed to prepare statement: %v", err)
        if err := dbTx.Rollback(); err != nil {
            LogError("Failed to rollback database transaction: %v", err)
        }
        return
    }
    defer stmt.Close()

    // Process batch with rollback protection
    for _, tx := range batch {
        txData, err := json.Marshal(tx)
        if err != nil {
            LogError("Failed to marshal transaction: %v", err)
            continue
        }

        _, err = stmt.Exec(tx.Hash, string(txData))
        if err != nil {
            LogError("Failed to insert/update transaction: %v", err)
            if err := dbTx.Rollback(); err != nil {
                LogError("Failed to rollback database transaction: %v", err)
            }
            return
        }
    }

    if err := dbTx.Commit(); err != nil {
        LogError("Failed to commit database transaction: %v", err)
        if err := dbTx.Rollback(); err != nil {
            LogError("Failed to rollback database transaction: %v", err)
        }
        return
    }
}
```

6.2.3 File System Safety

The system implements careful file system management:

1. **Path Validation**:
   - Sanitizes database file paths
   - Prevents directory traversal attacks
   - Ensures operations stay within config directory

2. **Permission Management**:
   - Database files created with 0600 permissions
   - Directories created with 0700 permissions
   - Prevents unauthorized access

3. **Resource Management**:
   - Proper file handle cleanup
   - Deferred statement closing
   - Memory leak prevention

The system also provides flexibility through configuration:

- Optional in-memory database for testing
- Configurable batch sizes
- Automatic database file creation

This comprehensive approach to database management ensures that:

- Transaction data is stored reliably
- System performance remains high under load
- Data integrity is maintained
- Recovery from errors is automatic
- Security is maintained at the storage layer

7 Network Synchronization

7.1 Full Sync Protocol

The sync process implements the following message structures:

```
type SyncRequest struct {
    LastHash      string
    LastTimestamp int64
}

type SyncResponse struct {
    Transactions []Transaction
    HasMore      bool
    NextHash     string
}
```

8 Security and Spam Prevention

The Hyyperlink network implements a comprehensive security and spam prevention system that combines reputation tracking, transaction validation, and economic incentives. This multi-layered approach ensures network health while remaining permissionless and decentralized.

8.1 Reputation System Architecture

The system employs a dual-reputation model that separately tracks both wallet and peer behavior. This separation allows for fine-grained control over network participation while maintaining flexibility for different types of actors.

8.1.1 Wallet Reputation

Wallet reputation is calculated using a formula that considers both transaction history and economic stake:

$$R_w = \begin{cases} R_{max} & \text{if } B \geq P \\ R_i \cdot (1 + \frac{B}{P}) & \text{otherwise} \end{cases} \tag{4}$$

where:

- R_w is the wallet's reputation score
- R_{max} is the maximum reputation (1.0)
- B is the wallet's current balance
- P is the poverty line threshold (10 HYY)
- R_i is the initial reputation score (1.0)

This formula creates several important security properties:

- Wealthy accounts automatically receive full trust, as they have economic stake
- New accounts receive a moderate initial trust level
- Trust scales with economic participation
- Malicious behavior results in reputation penalties

The implementation balances security with usability:

```
func (n *Node) GetWalletReputation(address string) float64 {
    balance := n.graph.State[address]
    if balance >= PovertyLine {
        return MaxReputation
    }

    reputation, exists := n.reputations[address]
    if !exists {
        balanceWeight := float64(balance) / float64(PovertyLine)
        return InitialReputation * (1 + balanceWeight)
    }
    return reputation
}
```

8.1.2 Transaction Validation Pipeline

Every transaction goes through a rigorous multi-stage validation process before being accepted into the network:

1. **Basic Structure Validation**:
   - Correct transaction format
   - Required fields present
   - Valid address formats
   - Minimum fee requirements
2. **Cryptographic Validation**:
   - Transaction hash verification
   - Signature authenticity
   - Mining proof validation (if applicable)
3. **Economic Validation**:
   - Balance sufficiency
   - Fee requirements
   - Double-spend prevention
4. **Reputation-Based Validation**:
   - Sender reputation check
   - Special rules for new accounts
   - Minimum amount requirements for first transactions

The validation pipeline is implemented as a series of checks:

```
func (g *Graph) ValidateTransaction(tx Transaction) error {
    // Stage 1: Basic validation
    if err := tx.ValidateBasic(); err != nil {
        return err
    }

    // Stage 2: Cryptographic validation
    if err := tx.ValidateCryptographic(); err != nil {
        return err
    }

    // Stage 3: Reference validation
    if err := g.ValidateReferences(tx); err != nil {
        return err
    }

    // Stage 4: State validation
    if err := g.ValidateState(tx); err != nil {
        return err
    }

    return nil
}
```

8.1.3 Anti-Spam Mechanisms

The system implements multiple layers of spam prevention:

1. **Economic Barriers**:
   - Mandatory transaction fees (0.0001 HYY)
   - All fees are burned, permanently reducing supply
   - Higher minimum amounts for first transactions (0.0001 HYY)
2. **Reputation Requirements**:
   - Minimum reputation threshold for transaction propagation
   - Reputation penalties for invalid transactions
   - Gradual trust building through successful transactions
3. **Network-Level Protection**:
   - Peer reputation tracking
   - Connection limits and pruning
   - Bandwidth allocation based on peer reputation

8.1.4 Attack Resistance

The combination of economic incentives and reputation tracking creates strong resistance to common attack vectors:

1. **Sybil Attacks**:
   - New accounts have limited privileges
   - Economic cost to create meaningful participation
   - Reputation building requires consistent good behavior

2. **Spam Attacks**:
   - Fee burning makes sustained spam expensive
   - Reputation loss for invalid transactions
   - Network-level filtering of low-reputation actors
3. **Eclipse Attacks**:
   - Persistent connections to validator nodes
   - Peer diversity requirements
   - Regular peer rotation and health checks

This comprehensive security approach ensures that:

- The network remains open while resistant to abuse
- Economic incentives align with network health
- Bad actors face increasing costs and decreasing capabilities
- Legitimate users can easily participate and build trust
- The system can automatically adapt to changing threat patterns

8.2 Transaction Filtering

The Hyyperlink network implements a transaction filtering system that acts as the first line of defense against spam and invalid transactions. This filtering happens before transactions enter the wider network, reducing unnecessary bandwidth usage and processing load.

8.2.1 Filtering Objectives

The transaction filtering system serves multiple purposes:

- Prevents obviously invalid transactions from propagating
- Enforces economic rules for network participation
- Protects against common spam patterns
- Provides special handling for new users
- Ensures efficient use of network resources

8.2.2 Core Filtering Rules

The system implements several key validation rules:

1. **Fee Validation**:
   - Every transaction must include a minimum base fee
   - Fees are burned to create deflationary pressure
2. **Reference Requirements**:
   - Transactions must reference existing transactions
   - Minimum reference count ensures DAG connectivity
   - References must be within valid time window
3. **New User Protection**:
   - First-time transactions face stricter requirements.
   - Mining proofs exempt from these restrictions

The filtering algorithm implements these rules in a systematic way:

Algorithm 2 Transaction Validation

```
if $tx.Fee < BaseFee$ then return ”fee too low”
end if
if $len(tx.References) < MinReferences$ then return ”insufficient references”
end if
if IsNewUser($tx.From$) and $tx.MiningProof = nil$ then
    $totalAmount \leftarrow 0$
    for each $change$ in $tx.Changes$ do
        $totalAmount \leftarrow totalAmount + change.Amount$
    end for
    if $totalAmount < MinimumFirstTransactionAmount$ then return ”first transaction amount too low”
    end if
end if
```

8.2.3 Special Cases and Exceptions

The filtering system includes special handling for certain transaction types:

1. **Mining Rewards**:
   - Bypass normal reference requirements
   - Must include valid mining proof
   - Still subject to fee requirements
2. **Validator Transactions**:
   - Priority processing
   - Relaxed reference requirements
   - Enhanced propagation rules
3. **High-Value Accounts**:
   - Accounts above poverty line get preferential treatment
   - Reduced filtering restrictions
   - Faster propagation through network

8.2.4 Impact on Network Health

The filtering system provides several key benefits:

- **Resource Efficiency**:
  - Early rejection of invalid transactions
  - Reduced network bandwidth usage
  - Lower processing overhead on nodes
- **Spam Prevention**:
  - Economic barriers to entry
  - Progressive difficulty for new accounts
  - Automatic filtering of dust transactions
- **Network Stability**:
  - Controlled transaction flow
  - Predictable resource usage
  - Self-regulating participation rules

This filtering approach ensures that the network can maintain high performance and security while remaining accessible to legitimate users. The combination of economic incentives, progressive restrictions, and special case handling creates a robust first line of defense against network abuse.

9 Transaction Model

Hyyperlink implements a flexible transaction model where each transaction represents one or more state changes to the network. While the basic use case is token transfer, the system is designed to support various types of state changes:

9.1 Transaction Structure

Each transaction contains:

- Source address (From)
- One or more state changes
- Network fee
- References to previous transactions
- Cryptographic signature

9.2 State Changes

State changes can represent:

- Basic token transfers
- NFT minting and transfers
- Token program instructions
- Smart contract state updates

The change structure is designed to be extensible:

```
type TransactionChange struct {
    To     string          // Target address
    Amount uint64          // Value for token transfers
    Data   interface{}     // Program instructions or metadata
}
```

9.3 Program Instructions

Transactions can carry program instructions that define operations like:

- Creating new tokens
- Minting NFTs
- Updating token properties
- Executing smart contract functions

This flexible model allows Hyyperlink to support a wide range of decentralized applications while maintaining the simplicity and efficiency of the core transaction system.

10 Account System

10.1 Account Creation

Hyyperlink accounts are derived directly from Ed25519 key pairs. The process is straightforward:

1. Generate a new Ed25519 key pair
2. Take the 32-byte public key
3. Encode it using Base58 encoding

The resulting Base58-encoded string serves as the account address. This process is implemented as:

```
func PublicKeyToAddress(publicKey ed25519.PublicKey) string {
    return base58.Encode(publicKey)
}

// Account creation example:
pubKey, privKey, _ := ed25519.GenerateKey(nil)
address := PublicKeyToAddress(pubKey)
```

10.2 Address Format

Addresses in Hyyperlink have the following properties:

- Fixed length of characters
- Base58 encoded (using digits and letters, excluding similar-looking characters)
- Derived directly from Ed25519 public keys
- No checksum or network prefix

This simple addressing scheme provides several benefits:

- Direct relationship between addresses and public keys
- No address generation overhead
- Efficient validation (just Base58 decode and length check)
- Human-readable format suitable for display and copying

Address validation is performed by checking the Base58 decoding and length:

```
func IsValidAddress(address string) bool {
    _, err := base58.Decode(address)
    return err == nil && len(address) == 44
}
```

11 Token Economics

11.1 Special Addresses

The system utilizes two special-purpose addresses for token management:

11.1.1 Token Grant Address

Hyyperlink network's special Token Grant address (TokenGrant1111111111111111111111111111111111) serves as the source of new token issuance. It has these unique properties:

- Source of all mining rewards
- Only issues tokens through valid mining proofs
- Zero initial balance
- Transactions from this address bypass normal validation rules

11.1.2 Burn Address

The Burn address (BURN1111111111111111111111111111111111111111) permanently removes tokens from circulation:

- Receives all transaction fees
- Cannot send tokens
- Tokens sent here are permanently removed from circulation
- Balance increases but never decreases

11.2 Token Units

Each Hyyperlink token (HYY) is divisible into 100,000,000 units:

- 1 HYY = 100,000,000 units
- Smallest unit = 0.00000001 HYY
- All internal calculations use integer units
- Display values are converted to decimal form

For example:

- 1.00000000 HYY = 100,000,000 units
- 0.00000001 HYY = 1 unit
- 123.45678900 HYY = 12,345,678,900 units

This level of divisibility ensures precise value transfer while maintaining integer-based calculations for maximum accuracy.

11.3 Supply Mechanics

The Hyyperlink token supply is governed by two opposing forces:

11.3.1 Supply Inflation

New tokens enter circulation through the mining process:

- Miners discover valid Ed25519 keys
- Token Grant address issues rewards based on difficulty
- Higher difficulties yield proportionally larger rewards
- Mining can occur in parallel without coordination

11.3.2 Supply Deflation

Tokens are removed from circulation through transaction fees:

- Every transaction requires a fee
- Fees are sent to the Burn address
- Burned tokens are permanently removed from supply
- Fee structure scales with transaction size

This dual mechanism creates a dynamic balance:

- Mining gradually increases the token supply
- Transaction activity gradually decreases the supply
- Network usage directly influences token economics
- Natural equilibrium between inflation and deflation

The system starts with zero tokens in circulation, relying on mining activity to bootstrap the initial supply. As network usage grows, transaction fees provide a deflationary pressure that helps maintain token value.

12 Conclusion

The Hyyperlink system presents a novel approach to digital currency, combining DAG-based transaction processing with reputation-based spam prevention and an innovative mining system. The architecture provides high throughput while maintaining security and decentralization through carefully designed incentive structures and validation mechanisms.