```
import asyncio
import asyncpg
import time
import json
import hashlib
import logging
import os
import uuid
from typing import List, Dict, Set
from collections import deque, defaultdict
import aiohttp
from pyin.client import LightningRpc
from ecdsa import SigningKey, SECP256k1
from heapq import heappush, heappop
from cryptography.fernet import Fernet
import numpy as np

# Configure logging for production monitoring
logging.basicConfig(level=logging.INFO, format="%(asctime)s - %(levelname)s -
%(message)s")
logger = logging.getLogger(__name__)

# Security and configuration constants
SHARD_COUNT = 22  # Adjusted for 50% Moore's Law increase by 2028 halving
TX_PER_BLOCK = 225  # Adjusted for 50% Moore's Law increase
TARGET_BLOCK_TIME = 4.0  # Unchanged
DIFFICULTY_ADJUSTMENT_PERIOD = 100  # Adjust difficulty every 100 blocks
DIFFICULTY_CHANGE_CAP = 0.1  # Max 10% difficulty change per adjustment
MAX_CHANNELS_PER_NODE = 100  # Limit channels per node
MIN_CHANNEL_BALANCE = 0.001  # Minimum balance to open a channel (BTC)
HTLC_MIN_TIMELOCK = 3600  # 1 hour in seconds
HTLC_MAX_TIMELOCK = 604800  # 1 week in seconds
MAX_RETRIES = 5  # Max transaction retries
RETRY_DELAY_BASE = 1  # Base delay for exponential backoff (seconds)
CROSS_SHARD_RATE_LIMIT = 0.1  # Minimum time between cross-shard transactions
(seconds)

# Encryption for sensitive data (e.g., balances)
ENCRYPTION_KEY = Fernet.generate_key()
cipher = Fernet(ENCRYPTION_KEY)

# Placeholder L1 wallet addresses (replace in production)
L1_TREASURY_WALLET = "bc1q_dev_wallet_placeholder"
L1_EMERGENCY_WALLET = "bc1q_emergency_dev_wallet_placeholder"
```

```
def generate_tx_id() -> str:
    """Generate a unique transaction ID."""
    return str(uuid.uuid4())

def generate_hash() -> str:
    """Generate a random hash for HTLC."""
    return hashlib.sha256(str(uuid.uuid4()).encode()).hexdigest()

def vrf_assign_shard(public_key: str, seed: str) -> int:
    """Assign a shard randomly using a VRF-like mechanism."""
    combined = public_key + seed
    hash_val = hashlib.sha256(combined.encode()).hexdigest()
    return int(hash_val, 16) % SHARD_COUNT

class PriceFeed:
    """Fetches real-time gold and BTC prices efficiently."""
    def __init__(self):
        self.gold_api_key = os.getenv("GOLD_API_KEY")
        if not self.gold_api_key:
            raise ValueError("GOLD_API_KEY environment variable is required")
        self.last_gold_price = None
        self.last_btc_price = None
        self.last_update = 0
        self.update_interval = 3600  # 1 hour

    async def get_prices(self) -> tuple[float, float]:
        """Fetch gold and BTC prices in a single batch request."""
        if time.time() - self.last_update < self.update_interval and self.last_gold_price and
self.last_btc_price:
            return self.last_gold_price, self.last_btc_price
        async with aiohttp.ClientSession() as session:
            gold_task = session.get("https://www.goldapi.io/api/XAU/USD",
headers={"x-access-token": self.gold_api_key})
            btc_task =
session.get("https://api.coingecko.com/api/v3/simple/price?ids=bitcoin&vs_currencies=usd")
            gold_resp, btc_resp = await asyncio.gather(gold_task, btc_task)
            gold_data = await gold_resp.json()
            btc_data = await btc_resp.json()
            self.last_gold_price = gold_data.get("price", 0)
            self.last_btc_price = btc_data.get("bitcoin", {}).get("usd", 0)
            self.last_update = time.time()
            return self.last_gold_price, self.last_btc_price
```

```
    async def get_node_fee(self) -> float:
        """Calculate node creation fee: 1% of 1 oz gold in BTC."""
        gold_price_usd, btc_price_usd = await self.get_prices()
        fee_usd = 0.01 * gold_price_usd
        return fee_usd / btc_price_usd

class Node:
    """Represents a network node."""
    def __init__(self, public_key: str, node_id: str, shard_id: int):
        self.public_key = public_key
        self.node_id = node_id
        self.shard_id = shard_id

class L2Block:
    """Represents an L2 block with optimized mining logic."""
    def __init__(self, transactions: List[Dict], previous_hash: str, shard_id: int, difficulty: int = 1):
        self.timestamp = time.time()
        self.transactions = transactions[:TX_PER_BLOCK]
        self.previous_hash = previous_hash
        self.shard_id = shard_id
        self.difficulty = difficulty
        self.nonce = 0
        self.hash = None

    async def mine_block(self):
        """Mine the block using PoW."""
        while True:
            data = f"{self.previous_hash}{json.dumps(self.transactions)}{self.nonce}".encode()
            self.hash = hashlib.sha256(data).hexdigest()
            if self.hash.startswith("0" * self.difficulty):
                break
            self.nonce += 1
            await asyncio.sleep(0)  # Yield control

class Channel:
    """Manages a Lightning Network payment channel with HTLC support."""
    def __init__(self, wallet_a, wallet_b, amount_a, amount_b, chain):
        self.wallet_a = wallet_a
        self.wallet_b = wallet_b
        self.balance_a = amount_a
        self.balance_b = amount_b
        self.state = "open"
        self.multisig_address = generate_multisig_address(wallet_a, wallet_b)
        self.chain = chain
```

```
        self.htlcs: Dict[str, Dict] = {}  # {htlc_id: {amount, hash, timelock, status}}

    def offer_htlc(self, amount, htlc_hash, timelock, from_wallet):
        """Offer an HTLC in the channel with timing constraints."""
        if timelock < time.time() + HTLC_MIN_TIMELOCK or timelock > time.time() +
HTLC_MAX_TIMELOCK:
            raise ValueError("HTLC timelock out of allowed range")
        if from_wallet == self.wallet_a:
            if self.balance_a < amount:
                raise ValueError("Insufficient funds in wallet_a")
            self.balance_a -= amount
        elif from_wallet == self.wallet_b:
            if self.balance_b < amount:
                raise ValueError("Insufficient funds in wallet_b")
            self.balance_b -= amount
        else:
            raise ValueError("Invalid wallet")
        htlc_id = generate_tx_id()
        self.htlcs[htlc_id] = {
            "amount": amount,
            "hash": htlc_hash,
            "timelock": timelock,
            "status": "offered",
            "from_wallet": from_wallet
        }
        return htlc_id

    async def settle_htlc(self, htlc_id, preimage):
        """Settle an HTLC with the preimage."""
        if htlc_id not in self.htlcs or self.htlcs[htlc_id]["status"] != "offered":
            raise ValueError("Invalid or settled HTLC")
        htlc = self.htlcs[htlc_id]
        if hashlib.sha256(preimage.encode()).hexdigest() != htlc["hash"]:
            raise ValueError("Invalid preimage")
        if time.time() > htlc["timelock"]:
            raise ValueError("HTLC timelock expired")
        if htlc["from_wallet"] == self.wallet_a:
            self.balance_b += htlc["amount"]
        else:
            self.balance_a += htlc["amount"]
        htlc["status"] = "settled"
        tx = {
            "type": "htlc_settle",
            "channel": self,
```

```
            "htlc_id": htlc_id,
            "preimage": preimage,
            "sender": htlc["from_wallet"].user_id,
            "signature": sign({"htlc_id": htlc_id, "preimage": preimage}, htlc["from_wallet"].privkey)
        }
        await self.chain.add_transaction(tx)

    async def refund_htlc(self, htlc_id):
        """Refund an HTLC if timelock expires."""
        if htlc_id not in self.htlcs or self.htlcs[htlc_id]["status"] != "offered":
            raise ValueError("Invalid or settled HTLC")
        htlc = self.htlcs[htlc_id]
        if time.time() <= htlc["timelock"]:
            raise ValueError("HTLC timelock not yet expired")
        if htlc["from_wallet"] == self.wallet_a:
            self.balance_a += htlc["amount"]
        else:
            self.balance_b += htlc["amount"]
        htlc["status"] = "refunded"
        tx = {
            "type": "htlc_refund",
            "channel": self,
            "htlc_id": htlc_id,
            "sender": htlc["from_wallet"].user_id,
            "signature": sign({"htlc_id": htlc_id}, htlc["from_wallet"].privkey)
        }
        await self.chain.add_transaction(tx)

    async def close(self):
        """Close the channel and settle on-chain."""
        if self.state != "open":
            raise ValueError("Channel is not open")
        self.state = "closed"
        tx = {
            "type": "channel_close",
            "channel": self,
            "sender": self.wallet_a.user_id,
            "amount_a": self.balance_a,
            "amount_b": self.balance_b,
            "signature": sign({"type": "channel_close", "amount_a": self.balance_a, "amount_b":
self.balance_b}, self.wallet_a.privkey)
        }
        await self.chain.add_transaction(tx)
```

```
class Wallet:
    """Manages user funds with fine-grained locking and Lightning support."""
    def __init__(self, user_id: str, balance: float, shard_id: int):
        self.user_id = user_id
        self.balance = balance
        self.shard_id = shard_id
        self.pending_transactions: Dict[str, float] = {}
        self.retrying_transactions: Dict[str, Dict] = {}
        self.balance_lock = asyncio.Lock()
        self.tx_lock = asyncio.Lock()
        self.recent_tx_timestamps = deque()
        self.last_cross_shard_tx_time = 0
        self.privkey = SigningKey.generate(curve=SECP256k1)
        self.pubkey = self.privkey.get_verifying_key()
        self.channels = []  # Track channels for exhaustion mitigation

    async def initiate_transaction(self, amount: float, receiver: str, receiver_shard: int, chain) ->
bool:
        """Initiate an on-chain transaction with retry flooding mitigation."""
        async with self.tx_lock:
            current_time = time.time()
            is_cross_shard = self.shard_id != receiver_shard
            if is_cross_shard and current_time - self.last_cross_shard_tx_time <
CROSS_SHARD_RATE_LIMIT:
                logger.warning(f"Cross-shard rate limit exceeded for shard {self.shard_id}")
                return False
            while self.recent_tx_timestamps and current_time - self.recent_tx_timestamps[0] >= 1:
                self.recent_tx_timestamps.popleft()
            if len(self.recent_tx_timestamps) >= 10:
                tx_id = generate_tx_id()
                self.retrying_transactions[tx_id] = {
                    "amount": amount, "retry_count": 0, "receiver": receiver, "dest_shard":
receiver_shard
                }
                logger.info(f"Transaction {tx_id} queued for retry")
                asyncio.create_task(self.retry_transaction(tx_id, chain))
                return False

            async with self.balance_lock:
                total_reserved = sum(self.pending_transactions.values()) + sum(tx["amount"] for tx in
self.retrying_transactions.values())
                if self.balance - total_reserved < amount:
                    logger.warning(f"Insufficient balance: {self.balance - total_reserved} < {amount}")
                    return False
```

```
            self.balance -= amount
            tx_id = generate_tx_id()
            self.pending_transactions[tx_id] = amount

        async with self.tx_lock:
            self.recent_tx_timestamps.append(current_time)
            if is_cross_shard:
                self.last_cross_shard_tx_time = current_time
            success = await self.send_to_network(tx_id, amount, receiver, receiver_shard,
is_cross_shard, chain)
            if success:
                del self.pending_transactions[tx_id]
                logger.info(f"Transaction {tx_id} completed successfully")
            else:
                async with self.balance_lock:
                    self.balance += amount
                    del self.pending_transactions[tx_id]
                logger.error(f"Transaction {tx_id} failed")
            return success

    async def send_to_network(self, tx_id: str, amount: float, receiver: str, dest_shard: int,
is_cross_shard: bool, chain) -> bool:
        """Send transaction with reduced propagation delay."""
        tx = {
            "tx_id": tx_id,
            "amount": amount,
            "sender": self.user_id,
            "receiver": receiver,
            "source_shard": self.shard_id,
            "dest_shard": dest_shard,
            "signature": sign({"tx_id": tx_id, "amount": amount}, self.privkey)
        }
        await chain.add_transaction(tx)
        await asyncio.sleep(0.02)
        return True

    async def retry_transaction(self, tx_id: str, chain):
        """Retry a transaction with exponential backoff."""
        tx = self.retrying_transactions[tx_id]
        retry_count = tx['retry_count']
        if retry_count >= MAX_RETRIES:
            logger.error(f"Transaction {tx_id} exceeded retry limit")
            del self.retrying_transactions[tx_id]
            return
```

```
        delay = RETRY_DELAY_BASE * (2 ** retry_count)  # Exponential backoff
        await asyncio.sleep(delay)
        tx['retry_count'] += 1
        success = await self.send_to_network(tx_id, tx['amount'], tx['receiver'], tx['dest_shard'],
self.shard_id != tx['dest_shard'], chain)
        if success:
            async with self.balance_lock:
                del self.pending_transactions[tx_id]
            logger.info(f"Transaction {tx_id} completed after retry")
        else:
            if tx['retry_count'] >= MAX_RETRIES:
                logger.error(f"Transaction {tx_id} failed after {MAX_RETRIES} retries")
                del self.retrying_transactions[tx_id]

    async def open_channel(self, receiver_wallet, amount, chain):
        """Open a Lightning payment channel with exhaustion mitigation."""
        if len(self.channels) >= MAX_CHANNELS_PER_NODE:
            raise ValueError("Maximum channels per node reached")
        if amount < MIN_CHANNEL_BALANCE:
            raise ValueError(f"Channel balance must be at least {MIN_CHANNEL_BALANCE}
BTC")
        async with self.balance_lock:
            if self.balance < amount:
                logger.warning(f"Insufficient balance to open channel: {self.balance} < {amount}")
                return None
            self.balance -= amount
        channel = Channel(self, receiver_wallet, amount, 0, chain)
        self.channels.append(channel)
        receiver_wallet.channels.append(channel)
        tx = {
            "type": "channel_open",
            "channel": channel,
            "sender": self.user_id,
            "signature": sign({"type": "channel_open", "amount": amount}, self.privkey)
        }
        await chain.add_transaction(tx)
        return channel

    async def send_payment(self, receiver_wallet, amount, chain, timelock=3600):
        """Send a payment through Lightning channels with onion routing."""
        path = await chain.route_payment(self, receiver_wallet, amount)
        if not path:
            logger.error("No valid payment path found")
            return False
```

```
        preimage = str(uuid.uuid4())
        htlc_hash = hashlib.sha256(preimage.encode()).hexdigest()
        timelock_deadline = time.time() + timelock
        onion_packet = create_onion_packet(path, preimage)  # Simplified onion routing
        for i in range(len(path) - 1):
            channel = chain.find_channel(path[i], path[i + 1])
            htlc_id = channel.offer_htlc(amount, htlc_hash, timelock_deadline, path[i])
        # Settle HTLCs with preimage (simplified)
        for i in range(len(path) - 1, 0, -1):
            channel = chain.find_channel(path[i - 1], path[i])
            await channel.settle_htlc(list(channel.htlcs.keys())[0], preimage)
        logger.info(f"Payment of {amount} BTC from {self.user_id} to {receiver_wallet.user_id}
completed")
        return True

    async def close_channel(self, channel, chain):
        """Close the channel."""
        await channel.close()

class CrossShardCoordinator:
    """Handles cross-shard transactions with asynchronous commit."""
    def __init__(self, chain):
        self.chain = chain
        self.active_transactions: Dict[str, Dict] = {}

    async def initiate_transaction(self, tx: Dict):
        """Execute a cross-shard transaction with timeouts and persistence."""
        tx_id = tx['tx_id']
        source_shard = tx['source_shard']
        dest_shard = tx['dest_shard']
        self.active_transactions[tx_id] = {'status': 'pending', 'shards': [source_shard, dest_shard],
'tx': tx}
        await self.save_tx_state(tx_id, 'pending')
        logger.info(f"Cross-shard tx {tx_id}: {source_shard} -> {dest_shard}")
        try:
            async with asyncio.timeout(10):
                await self.chain.lock_balance(tx['sender'], tx['amount'], source_shard)
                await self.chain.apply_transaction(tx, dest_shard)
                await self.chain.unlock_balance(tx['sender'], tx['amount'], source_shard)
                self.active_transactions[tx_id]['status'] = 'committed'
                await self.save_tx_state(tx_id, 'committed')
                logger.info(f"Cross-shard tx {tx_id} completed")
        except asyncio.TimeoutError:
            logger.error(f"Cross-shard tx {tx_id} timed out")
```

```
            self.active_transactions[tx_id]['status'] = 'aborted'
            await self.save_tx_state(tx_id, 'aborted')
        finally:
            del self.active_transactions[tx_id]

    async def save_tx_state(self, tx_id: str, status: str):
        """Persist transaction state for recovery."""
        async with self.chain.db_pool.acquire() as conn:
            await conn.execute(
                "INSERT INTO tx_states (tx_id, status) VALUES ($1, $2) ON CONFLICT (tx_id) DO
UPDATE SET status = $2",
                tx_id, status
            )

class L2BitcoinChain:
    """Core L2 chain with optimized operations and full Lightning support."""
    def __init__(self):
        self.shard_count = SHARD_COUNT
        self.price_feed = PriceFeed()
        self.coordinator = CrossShardCoordinator(self)
        self.pending_transactions = {i: asyncio.Queue() for i in range(self.shard_count)}
        self.shards = {i: [] for i in range(self.shard_count)}
        self.db_pool = None
        self.confirmed_balances: Dict[str, float] = {}
        self.shard_difficulties = {i: 1 for i in range(self.shard_count)}
        self.mining_times = {i: [] for i in range(self.shard_count)}  # Track mining times for difficulty
adjustment
        self.nodes = {}
        self.device_to_node = {}
        self.ip_to_nodes = {}
        self.shard_to_nodes = {i: set() for i in range(self.shard_count)}
        self.attack_mitigation_pool = 0.0
        self.max_wallet_balance = None
        self.node_fee = None
        self.channels = {}  # {multisig_address: Channel}
        self.network_graph = defaultdict(list)  # {wallet_id: [(neighbor_wallet_id, channel)]}
        self.network_seed = "initial_seed"  # Placeholder for network-wide seed
        self.lightning = LightningRpc(os.getenv("LIGHTNING_RPC_PATH")) if
os.getenv("LIGHTNING_RPC_PATH") else None

    async def init_db(self):
        """Initialize database with retry and optimized pool."""
        RETRY_DELAY = 1  # Seconds
        for attempt in range(MAX_RETRIES):
```

```
        try:
            self.db_pool = await asyncpg.create_pool(os.getenv("DB_DSN"), min_size=5,
max_size=20)
            async with self.db_pool.acquire() as conn:
                await conn.execute("""
                    CREATE TABLE IF NOT EXISTS balances (user_id TEXT PRIMARY KEY,
balance BYTEA);
                    CREATE TABLE IF NOT EXISTS blocks (
                        shard_id INTEGER, block_id INTEGER, timestamp REAL, hash TEXT
PRIMARY KEY, data JSONB,
                        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
                    );
                    CREATE TABLE IF NOT EXISTS tx_states (tx_id TEXT PRIMARY KEY, status
TEXT);
```