1 – Introduction

1.1 – The Birth of FactorialCoin

FactorialCoin was born in 2017 by Domero Software as an effort to create a revolutionary solution designed to tackle the main twelve principal cryptocurrency core challenges at the system level. The goal was to deliver a simplistic set of tools and solutions that create a more dynamic, economic, and free crypto ecosystem. This ecosystem would benefit all communities with more dynamic trading and research goals using cpu power, making it suitable for a wide range of communities and research institutions through its comprehensive tools and solutions.

The initial Factorial-Community-Coin (FCC) system includes the Coinbase-server, Nodelist-Service, Nodes, Explorer, Wallet, and Miner system, all of which are entirely written in Perl. The base public versions of the Node, Explorer, Miner, and Wallet also feature browser interfaces, making them accessible for Perl-based applications on Windows, Mac, and Linux. Additionally, iPhone and Android versions are currently in development. This manual is particularly relevant as these systems are not solely designed for Perl-based applications but are widely used and applicable. It aims to enable others to apply their talents to the design and implementation of new wallet and miner applications. This is especially crucial as we move into the next phase of connecting our Factorial Community Coin network to exchanges, which will require the implementation of our wallet system.

1.2 – Overview of FactorialCoin

The base systems of FactorialCoin comprise a Coinbase-Server, a Nodelist-Service, Decentralized-Nodes, Decentralized-Wallets, Decentralized-Miners, and an Explorer-API-Service for Wallets.

- **Coinbase-Server**: This server handles fee payout transactions every Sunday night at 0:00 AM FCC-Time for nodes that have processed transactions. It also manages coinbase transactions for all winning mining solutions and provides software updates for running nodes. These updates are necessary when new block types are created for the block-and-seal-chain model, extending the applicability of blockchain technology within FactorialCoin’s framework.
- **Nodelist-Service**: This service supplies Nodes, Wallets, Miners, and Explorers with online core nodes for their respective tasks. It ensures that each component can function efficiently by connecting to the necessary resources.
- **Decentralized-Nodes**: These nodes maintain the ledger and facilitate wallet balance checks and the creation and validation of new transactions. They interact with wallet leaves, signing transactions that are then propagated through the network of core nodes. The nodes achieve full consensus in a few rounds through a neural network and mathematical voting principle, adding processed wallet, coinbase, and fee transactions into the ledger. They also act as intermediaries between Miners and the Coinbase-Server, earning a coinbase fee for each miner who wins a coinbase transaction using their services. Once a transaction is successfully signed, it can be considered paid by the sender, awaiting network processing. If

the transaction is valid and not a double-spend attempt, it becomes spendable by the receiver upon processing.

- **Decentralized-Miners**: Miners work to find solutions to problems, sending these solutions to the coinbase server via the node they are connected to. When a mining round is lost, miners restart the process. Upon finding a solution, miners encrypt their wallet with the solution and send it to the node, which then forwards it to the coinbase server. The node informs the miner of the success or failure of the transaction. Successful solutions result in the coinbase server creating the coinbase transaction for the nodes to process into the ledger. Mining wins are prioritized in voting rounds, ensuring that both the miner and node are directly rewarded for their solutions.
- **Explorer-API-Service for Wallets**: This service provides access to wallet information and transaction history, allowing users to explore and interact with the blockchain data related to their wallets. And it also provides a http site for general ledger browser functionality to browse through the seal chain of ledger blocks.

These systems collectively create a robust and dynamic ecosystem for FactorialCoin, addressing key challenges in cryptocurrency and fostering a more versatile and efficient environment for transactions and mining.

1.3 – Purpose of this Manual

This manual aims to provide the reader with a foundational understanding of the FactorialCoin Interactive Systems and best practice principles within the network of applicable Wallets and Miners. It outlines the basic implementation information and context needed to build new wallet applications and mining solutions.

The structure of the manual is designed to guide the reader from understanding the basic predefined elements to the full application implementation. This approach ensures that by the end of the manual, the reader will have a comprehensive understanding of how to implement their wallet and/or miner application within their own working environment, while also gaining a holistic view of the FactorialCoin base system for which they are writing their implementation.

Key objectives of this manual include:

1. **Understanding FactorialCoin Systems**: Providing an overview of the various components of the FactorialCoin ecosystem, including the Coinbase-Server, Nodelist-Service, Decentralized-Nodes, Wallets, Miners, and the Explorer-API-Service.
2. **Implementation Guidance**: Offering step-by-step instructions and best practices for setting up and managing wallet and miner applications.
3. **Best Practices**: Highlighting the principles and best practices to ensure efficient, secure, and effective implementation and interaction with the FactorialCoin network.
4. **Holistic View**: Enabling developers to gain a complete understanding of how different components of the FactorialCoin system interact and integrate, fostering a deeper comprehension necessary for effective application development.

By following the guidance provided in this manual, developers will be equipped to contribute to the FactorialCoin ecosystem with their own applications, enhancing the versatility and functionality of the network.

2 – Nodelist Handeling

2.1 – Wallet and Miner Node-List Handling

To obtain the nodelist for FactorialCoin (FCC) or PTTP, you can make a simple HTTP SSL GET request to the respective URLs. The output is a space-separated list of `[ip]:[port]` pairs that represent the available nodes. These nodes can be used to connect your WebSocket client to a node.

FCC Nodelist URL

```
https://factorialcoin.nl:5151/?nodelist
```

Example Output:

```
141.138.137.123:7050 82.170.191.248:7050 217.62.128.84:7050 95.170.68.44:7051 87.253.131.187:7050
37.97.233.157:7050 95.170.68.44:7050 37.97.233.157:7051 85.148.8.48:7050 149.210.194.88:7050
31.187.148.85:7050 84.26.166.200:7050 82.170.191.248:7051 87.253.131.187:7051
```

PTTP Nodelist URL

```
https://factorialcoin.nl:9611/?nodelist
```

Example Output:

```
95.170.68.44:9369 149.210.194.88:9633 37.97.233.157:9634 37.97.233.157:9633 141.138.137.123:9633
31.187.148.85:9369
```

A nodelist consists of a list of `[ip]:[port]` pairs separated by a space character. By splitting the output using the space character, you get the list of online core nodes. Each element can be used to connect your WebSocket client to a node.

The list is randomly ordered, and to distribute the network load, you should randomly pick nodes from the list for your transactions, balance checks, or mining calls. This approach ensures that multiple requests are spread across the network, enhancing load balancing. You can work through the list randomly until the last node, then start over with a new round of randomly using nodes to ensure all nodes are utilized before reusing any node.

This approach ensures efficient distribution of network load and helps maintain a balanced and robust FactorialCoin network.

2.2 – Backing Up and Recycling the online core nodes

To ensure efficient operation and minimize unnecessary network load, it is crucial to buffer or back up the nodelist. This practice helps to limit the number of requests to the nodelist service, especially when dealing with potentially millions of users. By caching the nodelist, you can speed up the

application’s access to online core nodes and efficiently manage the connections throughout the operation of a wallet or miner.

Here are the steps and strategies for backing up and recycling the online core nodes:

1. **Buffering the Nodelist**:
    - Store the fetched nodelist in memory or on disk to minimize repeated requests to the nodelist service.
    - Refresh the nodelist periodically or when it becomes too small to ensure you are working with online core nodes.
2. **Recycling Nodes**:
    - Implement a mechanism to mark nodes as offline when they become unreachable.
    - Keep track of node performance and uptime to filter out weaker nodes and prioritize those with higher reliability.
3. **Offline Backups**:
    - Maintain an offline backup of the nodelist to connect directly to nodes without requesting a new nodelist at startup.
    - Use the backup list to quickly establish initial connections and then update the list as necessary.
4. **Optimizing Node Usage**:
    - Randomly select nodes for transactions to distribute the load evenly across the network.
5. **Handling Large Networks**:
    - Plan for scalability by considering mechanisms to limit the size of the nodelist returned by the service as the network grows beyond 500 nodes.

This approach ensures efficient management and recycling of the nodelist, promoting reliable performance for wallet and miner applications across the FactorialCoin network.

3 – Node-leaf Connection Handeling

3.1 – Setting Up Node-leaf Websocket Connection

To set up a WebSocket connection to a FactorialCoin node, you need to implement a class API for handling the non-blocking WebSocket communication. This includes connecting to the node, managing incoming and outgoing events, and processing JSON messages.

When your client connects, the node will initiate communication with a "hello" message. Your client should respond with an "identify" message to keep the connection open and start using the wallet or miner functionalities.

Important Notes:

- A node's unprocessed message limit is 5 messages before it will kick the connection on the 6th message send, so manage your message queue carefully. Wait atleast for a return before you send your next message to the node.
- For wallet transactions, treat the connection like a MySQL query connection sequence, completing the transaction and/or balance call before disconnecting, to leave port space for other connections to the nodes.
- To avoid multiple in-block spending errors, use the same node for transactions from the same wallet.
- Multiple transactions from different wallets can be processed concurrently by multiple nodes without issues.
- Handle connection errors by reconnecting to a new node and resuming or redoing your process as needed.

WebSocket Initial Hello Message from Node

```
json

{
  "command": "hello",
  "host": "[node-ip]",
  "port": "[node-port]",
  "version": "[fcc-ledger-version]"
}
```

3.2 Setting Up Wallet Node-Leaf

After establishing the WebSocket connection and receiving the "hello" message from the node, you need to send an "identify" message to start using wallet-specific commands.

Identify Message for Wallet Node-Leaf

```
json

{
  "command": "identify",
  "type": "leaf",
  "version": "[fcc-ledger-version]"
}
```

3.3 Setting Up Miner Node-Leaf

For a miner node-leaf, the process is similar, but you identify as a "miner".

Identify Message for Miner Node-Leaf

```
json

{
  "command": "identify",
  "type": "miner",
  "version": "[fcc-ledger-version]"
}
```

4 – Wallet Creation and Management

4.1 - Steps to Create a New FCC Wallet Object

Creating a new FactorialCoin (FCC) wallet involves several key steps. The following sections describe the detailed process of creating a new wallet object, validating it, and generating the necessary cryptographic keys. This guide is intended for developers looking to implement wallet creation functionality for the FCC network in their applications.

1. Wallet Structure

The FCC wallet address structure consists of a specific format:

- **Offset 0-1:** '51' - FCC identifier (or '11' for PTTP)
- **Offset 2-65:** Public hashkey
- **Offset 66-67:** Checksum, calculated by XORing ASCII values from offset 0-65, must be 0.

2. Create XOR Table

A table of XOR values is created to help with checksum calculation.

```
perl

# Initialize the array @WXOR as empty
my @WXOR = ();

# Call the createtable subroutine to populate @WXOR with XOR values
createtable();

# Subroutine createtable to generate XOR values and store them in @WXOR
sub createtable {
    # Initialize an array @l to store ASCII values from 0 to 9 and A to F
    my @l = ();

    # Populate @l with ASCII values from 0 to 9
    for (my $c = 0; $c < 10; $c++) {
        push @l, ord($c);
    }

    # Populate @l with ASCII values from A to F
    for (my $c = 'A'; $c le 'F'; $c++) {
        push @l, ord($c);
    }

    # Iterate through each element in @l
    foreach my $m (@l) {
        foreach my $n (@l) {
            # Calculate XOR value between $m and $n
            my $xor_value = $m ^ $n;

            # Push a hash reference into @WXOR with 'add' as concatenated characters of $m and $n
            # and 'value' as the calculated XOR value
            push @WXOR, { add => chr($m) . chr($n), value => $xor_value };
        }
    }
}
```

Explanation:

Initialization and Function Call:

- **@WXOR Initialization**: Declares `@WXOR` as an empty array.
- **createtable() Call**: Invokes the `createtable()` subroutine to populate `@WXOR`.

`createtable`:

- **Purpose**: Generates a table of XOR values and stores them in `@WXOR`.
- **Steps**:
  - **Initialization**: Initializes `@l` to store ASCII values from '0' to '9' and 'A' to 'F'.
  - **Population of @l**:
    - Loops through ASCII values '0' to '9' and pushes their corresponding ordinals into `@l`.
    - Loops through ASCII values 'A' to 'F' and pushes their corresponding ordinals into `@l`.
  - **XOR Calculation**:
    - Nested loops iterate over each element `$m` and `$n` in `@l`.
    - Calculates the XOR (`$m ^ $n`) and stores the result in `$xor_value`.
  - **Hash Construction**:
    - Constructs a hash reference `{ add => chr($m) . chr($n), value => $xor_value }`.
    - Pushes this hash reference into `@WXOR`, where `'add'` is the concatenation of characters represented by `$m` and `$n`, and `'value'` is the XOR result.

3. Generate Cryptographic Key Pair

Use the `Crypt::Ed25519` module to generate a public and private key pair.

```
perl

# Generate a new key pair using Crypt::Ed25519 module
my ($pubkey, $privkey) = Crypt::Ed25519::generate_keypair;

# Convert the public key to hexadecimal format using octhex subroutine
my $pubhex = octhex($pubkey);
```

Explanation:

Crypt::Ed25519::generate_keypair:

- **Purpose**: This function generates a new cryptographic key pair using the Ed25519 elliptic curve algorithm provided by the `Crypt::Ed25519` Perl module.
- **Return Values**: It returns two values:
  - `$pubkey`: The public key generated by the algorithm.
  - `$privkey`: The corresponding private key.

octhex($pubkey):

- **Purpose**: The `octhex` subroutine converts binary data (likely the public key `$pubkey`) into a hexadecimal string representation.

- **Usage**: Takes `$pubkey` as input and returns its hexadecimal representation.

4. Create Wallet Address

The wallet address is created using the public key. This includes hashing the public key and calculating a checksum.

```
perl

sub createwalletaddress {
    my ($pubkey) = @_;

    # Compute SHA-256 hash of the public key
    my $pubhash = securehash($pubkey);

    # Initial XOR value for FCC (ASCII '5' XOR ASCII '1')
    my $xor = ord('5') ^ ord('1');

    # Calculate XOR checksum based on pubhash
    for (my $c = 0; $c < 64; $c++) {
        $xor ^= ord(substr($pubhash, $c, 1));
    }

    my $checksum = "";

    # Lookup the correct checksum value from @WXOR based on XOR result
    foreach my $try (@WXOR) {
        if (($try->{value} ^ $xor) == 0) {
            $checksum = $try->{add};
            last;
        }
    }

    # Construct the wallet address format for FCC
    return '51' . $pubhash . $checksum;
}
```

Explanation:

`createwalletaddress`:

- **Input**: Takes `$pubkey` as a parameter, which is the public key of the wallet.
- **Compute Hash**: Computes the SHA-256 hash (`$pubhash`) of the `$pubkey`.
- **Initial XOR Value**: Sets an initial XOR value (`$xor`) to differentiate the checksum calculation. For FactorialCoin (FCC), it starts with `ord('5') ^ ord('1')`.
- **XOR Calculation Loop**:
  - Iterates through the first 64 characters of `$pubhash`.
  - XORs each character of `$pubhash` with the current `$xor` value.
- **Checksum Lookup**:
  - Searches through the `@WXOR` array to find the correct checksum (`$checksum`) based on the XOR result (`$xor`).
  - If a match is found (`($try->{value} ^ $xor) == 0`), sets `$checksum` to `$try->{add}` and breaks out of the loop.
- **Return Value**: Constructs and returns the wallet address in the format `'51' . $pubhash . $checksum` for FactorialCoin (FCC).

5. Validate Wallet

Validation ensures the wallet address conforms to the expected format and checksum.

```
perl

sub validwallet {
    my ($wallet) = @_;

    # Return false if no wallet provided
    if (!$wallet) { return 0 }

    # Convert wallet to uppercase for consistency
    $wallet = uc($wallet);

    # Check if wallet length is exactly 68 characters
    if (length($wallet) != 68) { return 0 }

    # Initial XOR value for FCC (ASCII '5' XOR ASCII '1')
    my $xor = ord('5') ^ ord('1');

    # Iterate through each character of the wallet string
    for (my $c = 2; $c < 68; $c++) {
        my $h = substr($wallet, $c, 1);

        # Validate character: must be hex digit (0-9, A-F)
        if ((($h ge '0') && ($h le '9')) || (($h ge 'A') && ($h le 'F'))) {
            $xor ^= ord($h);  # XOR current character with $xor
        } else {
            return 0;  # Return false if character is not a valid hex digit
        }
    }

    # Return true if final XOR value is zero (valid wallet)
    return ($xor == 0);
}
```

Explanation:

`validwallet`:

- **Input**: Takes `$wallet` as a parameter, which should be a hexadecimal string representation of a wallet.
- **Initial Check**: Immediately returns false (`0`) if `$wallet` is not defined or empty.
- **Uppercase Conversion**: Converts `$wallet` to uppercase to ensure consistency in case.
- **Length Check**: Returns false (`0`) if the length of `$wallet` is not exactly 68 characters.
- **Initial XOR Value**: Sets an initial XOR value (`$xor`) to differentiate the checksum calculation.
- **Character Validation Loop**:
    - Iterates through each character (`$h`) of the wallet string starting from index `2` (skipping the first two characters).
    - Validates that each character is a valid hexadecimal digit (`0-9` or `A-F`).
    - XORs the ASCII value of each valid character with the current `$xor` value.
    - If a character is not valid (not a hex digit), the subroutine immediately returns false (`0`).
- **Final Check**: Returns true (`1`) if the final value of `$xor` equals `0`, indicating the wallet is valid according to the checksum calculation.

6. Creating the Wallet Object

The new wallet object includes the public key, private key, wallet address, and optional name.

```
perl

use Crypt::Ed25519;
```

```
sub newwallet {
    my ($name) = @_;

    # Set default name if not provided
    if (!$name) { $name = "[ No name ]" }

    # Generate Ed25519 key pair
    my ($pubkey, $privkey) = Crypt::Ed25519::generate_keypair;

    # Convert keys to hexadecimal representation
    my $pubhex = octhex($pubkey);
    my $privhex = octhex($privkey);

    # Create wallet data structure
    my $wallet = {
        pubkey  => $pubhex,                         # Public key for sharing
        privkey => $privhex,                        # Private key for signing
        wallet  => createwalletaddress($pubhex),    # Wallet address for transactions
        name    => $name                            # Name of the wallet
    };

    # Bless the wallet reference as an object
    bless($wallet);

    # Return the constructed wallet object
    return $wallet;
}
```

Explanation:

`newwallet`:

- **Input**: Takes `$name` as an optional parameter to assign a name to the wallet. If not provided, defaults to `"[ No name ]"`.
- **Generate Keys**: Uses `Crypt::Ed25519::generate_keypair` to generate a new Ed25519 key pair consisting of a public key (`$pubkey`) and a private key (`$privkey`).
- **Hexadecimal Conversion**: Converts the generated keys (`$pubkey` and `$privkey`) into their hexadecimal representations using the `octhex` subroutine.
- **Create Wallet Object**:
    - Constructs a hash reference (`$wallet`) with keys:
        - `pubkey`: Holds the hexadecimal representation of the public key, used for sharing.
        - `privkey`: Holds the hexadecimal representation of the private key, used for signing data.
        - `wallet`: Calls `createwalletaddress($pubhex)` to generate a wallet address based on the public key hexadecimal representation.
        - `name`: Stores the provided or default name for the wallet.
- **Object Blessing**: Blesses the `$wallet` reference as an object to enable object-oriented programming features.
- **Return**: Returns the constructed `$wallet` object.

7. Secure Hash Function

A secure hash function is used for generating wallet addresses.

```
perl

sub securehash {
    my ($code) = @_;
```

```
    # Check if $code is defined
    die "FCC.Global.SecureHash: No Code given to hash!" if (!$code);

    # Calculate SHA-512 hash of $code and then SHA-256 hash of the result
    my $sha512_hash = sha512_hex($code);
    my $sha256_hash = sha256_hex($sha512_hash);

    # Convert the final hash to uppercase
    return uc($sha256_hash);
}
```

Explanation:

`securehash`:

- **Input**: Takes `$code`, which is the string to be hashed.
- **Error Checking**: Throws an error and terminates the script with `die` if `$code` is not defined (`!$code`).
- **Hash Calculation**:
  - Calculates the SHA-512 hash of `$code` using `sha512_hex()`.
  - Calculates the SHA-256 hash of the SHA-512 hash obtained in the previous step using `sha256_hex()`.
- **Uppercase Conversion**: Converts the resulting SHA-256 hash to uppercase using `uc()`.
- **Output**: Returns the uppercase hexadecimal representation of the double-hashed `$code`.

8. Hexadecimal Conversion Functions

Functions for converting between octal and hexadecimal formats are provided.

```
perl

sub octhex {
    my ($key) = @_;

    my $hex = "";
    # Convert each character in $key to its hexadecimal representation
    for (my $i = 0; $i < length($key); $i++) {
        my $c = ord(substr($key, $i, 1));
        $hex .= sprintf('%02X', $c);
    }

    return $hex;
}

sub hexoct {
    my ($hex) = @_;

    my $key = "";
    # Convert each pair of hexadecimal digits in $hex back to characters
    for (my $i = 0; $i < length($hex); $i += 2) {
        my $h = substr($hex, $i, 2);
        $key .= chr(hex($h));
    }

    return $key;
}
```

Explanation:

`octhex`:

- **Input**: Takes `$key`, which is a string of characters to convert to hexadecimal.

- **Loop**: Iterates through each character (`$c`) of `$key`.
    - Converts each character to its ASCII value using `ord()`.
    - Formats the ASCII value as a two-digit hexadecimal string using `sprintf('%02X', $c)`.
    - Concatenates each formatted hexadecimal string to `$hex`.
- **Output**: Returns the concatenated `$hex` string containing the hexadecimal representation of each character in `$key`.

`hexoct`:

- **Input**: Takes `$hex`, which is a string of hexadecimal digits to convert back to characters.
- **Loop**: Iterates through `$hex` in steps of two characters (`$i += 2`).
    - Extracts each pair of hexadecimal digits (`$h`) from `$hex`.
    - Converts `$h` from hexadecimal to its ASCII character representation using `chr(hex($h))`.
    - Concatenates each converted character to `$key`.
- **Output**: Returns the concatenated `$key` string containing the characters decoded from `$hex`.

Summary

This chapter outlined the steps and provided code snippets to create a new FCC wallet. Developers should be able to use these details to implement wallet creation functionality in their own applications, ensuring compatibility with the FactorialCoin network.

4.2 - Securing the Wallet

Overview

Securing a wallet is a critical step in the process of wallet management. It ensures that the wallet's private keys and sensitive information are protected from unauthorized access. This section will describe the methods and processes for securing a FactorialCoin (FCC) wallet, including encryption techniques and password protection.

Steps to Secure a Wallet

1. Encrypting Wallet Data

The wallet data, including private keys and sensitive information, should be encrypted using a secure hashing mechanism. The `fccencode` function is used to achieve this encryption.

```
perl

sub fccencode {
    my ($data, $password) = @_;

    # Generate secure hashes of the password and its reverse
    my $h1 = securehash($password);
    my $h2 = securehash(scalar reverse $password);

    my $pos = 0;
```

```
    my $todo = length($data);
    my $dpos = 0;
    my $coded = "";

    # Encode data by XORing with hashed password values
    while ($dpos < $todo) {
        # Determine offset using $h2 hash
        my $get = $HP->{substr($h2, $pos, 1)};
        $pos += $get;
        $pos %= 64;
        $pos = 0 if $pos == 63;

        # Calculate code using $h1 hash
        my $code =
            ($HP->{substr($h1, $pos, 1)} << 4) +
            $HP->{substr($h1, $pos + 1, 1)};

        # XOR operation between data character and code
        my $tocode = ord(substr($data, $dpos, 1));
        $coded .= chr($code ^ $tocode);

        $dpos++;
    }

    # Convert encoded data to hexadecimal format
    return octhex($coded);
}
```

Explanation:

- **Input:** The data to be encoded and a password.
- **Hash Generation:** Secure hashes of the password and its reverse are generated.
- **Encoding Loop:** Each character of the data is XORed with a value derived from the hashes.
- **Output:** The encoded data is converted to a hexadecimal format.

2. Password Protection

When saving or loading wallet data, it is essential to use a password to protect the encrypted information. This ensures that only users with the correct password can access the wallets private keys.

Example: Adding new Wallet Object and Saving a Wallet with Password Protection

```
perl

sub savewallet {
    my ($wallet, $password) = @_;

    # Validate that $wallet is an instance of FCC::wallet
    if (ref($wallet) ne "FCC::wallet") {
        die "FCC::wallet::savewallet - Wallet given is not a FCC blessed wallet";
    }

    # Load existing wallets with the provided password
    my $wlist = loadwallets($password);

    # Check if loading returned an error due to incorrect password
    if (($#{$wlist} == 0) && ($wlist->[0]{error})) {
        die "FCC::wallet::savewallet - Adding wallet with wrong password";
    }

    # Add the new wallet to the loaded wallet list
    push @{$wlist}, $wallet;

    # Save the updated wallet list including the new wallet
    savewallets($wlist, $password);
}
```

Explanation:

- **Validation**: Checks if the provided `$wallet` object is an instance of `FCC::wallet`. If not, it throws an error indicating that the wallet type is incorrect.
- **Loading Wallets**: Calls the `loadwallets` subroutine to load existing wallets using the provided `$password`.
- **Handling Errors**: Checks if `loadwallets` returned an error (indicating an incorrect password scenario) and throws an appropriate error message if so.
- **Adding Wallet**: Pushes the validated `$wallet` object into the `$wlist` array.
- **Saving Wallets**: Calls the `savewallets` subroutine to save the updated `$wlist` array (including the newly added wallet) with the provided `$password`.

4.3 - Backing Up and Exporting Wallet Information

Overview

Backing up wallet information is crucial to prevent data loss. Users can export their wallet data to json encoded files or QR codes for secure storage and easy recovery. This section will detail the processes for backing up and restoring wallet information, including file-based and QR-based methods.

Steps to Backup and Restore Wallet Information

1. Exporting Wallet Information

Wallet data can be exported to a file for backup. This file contains all the necessary information, including private keys, public keys, and contact-book information.

Example: Saving Wallets to a File

```
perl

sub savewallets {
    my ($wlist, $password) = @_;

    # Initialize an empty string for encryption seed
    my $enc = "";

    # Generate encryption seed and hash if password is provided
    if ($password) {
        my $seed = "";
        for (my $i = 0; $i < 8; $i++) {
            # Generate random hex characters for seed
            $seed .= hexchar(int rand(16));
        }
        # Create encryption key
        $enc = $seed . securehash($seed . $COIN . $password);
    }

    # Initialize an empty array reference for processed wallets
    my $wcl = [];

    # Iterate through each wallet in the provided list
    foreach my $w (@$wlist) {
        my $wallet = {};

        # Copy name, wallet, pubkey, and privkey fields if they exist
        if ($w->{name}) {
            $wallet->{name} = $w→{name};
        }
        if ($w->{wallet}) {
```

```
            $wallet->{wallet} = $w->{wallet};
        }
        if ($w->{pubkey}) {
            # Encode pubkey using FCC encoding function if password is provided,
            # otherwise keep as is
            $wallet->{pubkey} = $password ?
                fccencode(hexoct($w->{pubkey}), $password) :
                $w->{pubkey};
        }
        if ($w->{privkey}) {
            # Encode privkey using FCC encoding function if password is provided,
            # otherwise keep as is
            $wallet->{privkey} = $password ?
                fccencode(hexoct($w->{privkey}), $password) :
                $w->{privkey};
        }

        # Handle types (private, public, contact) and their debit information
        for my $type ('private', 'public', 'contact') {
            if (ref($w->{$type})) {
                # Copy type information
                $wallet->{$type} = $w->{$type};

                if (ref($wallet->{$type}{Debit})) {
                    # Encode debit pubkeys and privkeys if password is provided,
                    # otherwise keep as is
                    for my $c (keys %{$wallet->{$type}{Debit}}) {
                        for my $d (@{$wallet->{$type}{Debit}{$c}}) {
                            if ($d->{pubkey}) {
                                $d->{pubkey} = $password ?
                                    fccencode(hexoct($d->{pubkey}), $password) :
                                    $d->{pubkey};
                            }
                            if ($d->{privkey}) {
                                $d->{privkey} = $password ?
                                    fccencode(hexoct($d->{privkey}), $password) :
                                    $d->{privkey};
                            }
                        }
                    }
                }
            }
        }

        # Push processed wallet into $wcl array
        push @$wcl, $wallet;
    }

    # Create or update the wallet file with encoded information
    gfio::create(
        "$WALLETDIR/wallet$FCCEXT",
        toJSON(
            {
                encoded => $enc,
                version => '2.1',
                wlist => $wcl
            }
        )
    );
}
```

Explanation:

- **Initialization**: Declares variables `$wlist` and `$password`.
- **Encryption Seed**: Generates a random seed and computes an encryption key (`$enc`) based on the provided password.
- **Processing Wallets**: Iterates through each wallet (`$w`) in the `$wlist`.
- Copies basic fields (`name`, `wallet`, `pubkey`, `privkey`) to `$wallet`.
- Encodes `pubkey` and `privkey` using `fccencode` if a password is provided.
- Handles `private`, `public`, and `contact` types and their `Debit` information:
    - Copies type information (`private`, `public`, `contact`).

    - Encodes `Debit` pubkeys and privkeys using `fccencode` if a password is provided.
- **File Creation**: Creates or updates the wallet file (`"$WALLETDIR/wallet$FCCEXT"`) with JSON-encoded content (`$enc`, `version`, `wlist`).

2. Importing Wallet Information

Users can import wallet information from a backup file. This process involves decrypting the file using the correct password and loading the wallet data.

Example: Loading Wallets from a File

```
perl

sub loadwallets {
    my ($password) = @_;

    # Initialize an empty array reference to hold wallet information
    my $wlist = [];

    # Check if the wallet file exists
    if (-e "$WALLETDIR/wallet$FCCEXT") {
        # Read and decode JSON content from the wallet file
        my $winfo = dec_json(gfio::content("$WALLETDIR/wallet$FCCEXT"));

        # Ensure the decoded content is a hash reference
        if (ref($winfo) eq 'HASH') {
            # If the wallet information is encoded, verify the password
            if ($winfo->{encoded}) {
                # Extract seed and hash from the encoded information
                my ($seed, $hash) = (substr($winfo->{encoded}, 0, 8), substr($winfo->{encoded}, 8));

                # Verify the password using a secure hash function
                if (securehash($seed . $COIN . $password) ne $hash) {
                    # Return error if password is invalid
                    return [ { error => 'invalid password' } ];
                }
            }

            # Assign the wallet list from decoded information
            $wlist = $winfo->{wlist};

            # Iterate through each wallet in the list
            foreach my $wallet (@$wlist) {
                # Bless the wallet object
                bless($wallet);
                # Set default name if not provided
                $wallet->{name} = "[ No name ]" if !$wallet→{name};

                # If wallet information is encoded, decode sensitive data using the password
                if ($winfo->{encoded}) {
                    # Encode public and private keys using FCC encoding function
                    if ($wallet->{pubkey}) {
                        $wallet->{pubkey} = fccencode(
                            hexoct($wallet->{pubkey}),
                            $password
                        );
                    }
                    if ($wallet->{privkey}) {
                        $wallet->{privkey} = fccencode(
                            hexoct($wallet->{privkey}),
                            $password
                        );
                    }

                    # Encode debit information in different types (private, public, contact)
                    for my $type ('private', 'public', 'contact') {
                        if (ref($wallet->{$type}{Debit})) {
                            for my $c (keys %{$wallet->{$type}{Debit}}) {
                                for my $d (@{$wallet->{$type}{Debit}{$c}}) {
                                    if ($d->{pubkey}) {
```

```
                                    $d->{pubkey} = fccencode(
                                        hexoct($d→{pubkey}),
                                        $password
                                    );
                                }
                                if ($d->{privkey}) {
                                    $d->{privkey} = fccencode(
                                        hexoct($d→{privkey}),
                                        $password
                                    );
                                }
                            }
                        }
                    }
                }
            }
        } else {
            # If the decoded content is not a hash reference, assign it directly to $wlist
            $wlist = $winfo;

            # Set default names for wallets if not provided
            foreach my $wallet (@$wlist) {
                $wallet->{name} = "[ No name ]" if !$wallet->{name};
            }
        }
    }

    # Return the list of wallets
    return $wlist;
}
```

Explanation:

1. **Subroutine Definition**: Defines a subroutine named `loadwallets` which takes a single argument `$password`.
2. **Initialization**: Initializes an empty array reference `$wlist` to store wallet information retrieved from the wallet file.
3. **File Existence Check**: Checks if the wallet file (`"$WALLETDIR/wallet$FCCEXT"`) exists.
4. **Read and Decode Wallet Information**:
    - Uses `gfio::content` to read the content of the wallet file.
    - Uses `dec_json` to decode the JSON content into Perl data.
5. **Handling Decoded Information**:
    - Checks if the decoded information (`$winfo`) is a hash reference.
    - If encoded, verifies the password using a secure hash (`securehash` function).
    - If password verification fails, returns an array reference containing an error message.
6. **Processing Wallets**:
    - Assigns the wallet list (`$wlist`) from the decoded information (`$winfo->{wlist}`).
    - Iterates through each wallet in `$wlist`.
    - Blesses each wallet object.
    - Sets a default name (`"[ No name ]"`) if the wallet name is not provided.
7. **Encoding Sensitive Information**:

- If wallet information is encoded:
    - Encodes `pubkey` and `privkey` fields using `fccencode` function and the provided `$password`.
    - Encodes debit information (`Debit` fields) in different types (`private`, `public`, `contact`).

8. **Fallback Handling**:
    - If `$winfo` is not a hash reference, directly assigns it to `$wlist`.
    - Sets default names for wallets if not provided.
9. **Return Statement**: Returns the populated `$wlist` containing wallet information, potentially with encoded sensitive data.

QR Code Backup

In a later stage and Edition we will add more information about the QR code protocols implemented by the FactorialCoin system. For mobile devices, wallet information can be exported and imported using QR codes. This allows users to back up and restore their wallet data using their device's camera. PC and Web Wallet interfaces can also export with QR codes. Even whole wallet files into a single Pdf containing all the wallet and contact data. Each QR code protected by one or more passwords and encoding levels.

Summary

This chapter detailed the processes for securing and backing up wallet information, including file-based and QR-based methods. Developers should implement these security measures to ensure that wallet data is protected and can be easily restored.

4.4 - Best Practices for Backup

Ensuring Wallet Security and Recoverability

Properly backing up your wallet information is crucial for maintaining the security and accessibility of your funds. By adhering to best practices, you can minimize the risk of data loss and ensure that your wallet can be restored in case of emergencies. Here are some essential guidelines to follow:

1. Regularly Backup Your Wallet Information

- **Frequency:** Make backups a routine part of your wallet management process. After any significant transaction or change to your wallet, create a new backup.
- **Automation:** Utilize software that can automate the backup process to ensure consistency and reduce the risk of human error.

2. Store Backups in Multiple, Secure Locations

- **Diversity:** Store your backups in various locations to protect against loss due to physical damage, theft, or technical failures. Consider both physical storage (e.g., USB drives, external hard drives) and cloud storage solutions.

- **Security:** Ensure that all storage locations are secure. For physical media, use safes or secure lockboxes. For cloud storage, use reputable providers with strong security measures.

3. Use Strong Passwords and Encryption for All Backups

- **Password Strength:** Choose strong, unique passwords for your backups. Avoid using easily guessable passwords or reusing passwords from other accounts.
- **Encryption:** Encrypt your backup files to add an extra layer of security. This ensures that even if someone gains access to your backup, they cannot read or use the data without the encryption key.

4. Test Your Backup and Restoration Procedures Periodically

- **Verification:** Regularly test your backups by restoring them to verify that they work correctly. This helps to identify any potential issues with the backup or restoration process.
- **Update Procedures:** Ensure that your backup and restoration procedures are up-to-date with the latest software and security practices. Make any necessary adjustments to improve the process.

Summary

By following these best practices, users can ensure that their wallet information is secure, accessible, and recoverable in case of emergencies. Regular backups, secure storage, strong passwords, and periodic testing form the foundation of a robust wallet backup strategy. These steps are essential for safeguarding your digital assets and ensuring peace of mind.

4.5 - Managing Wallet Functionalities

Overview

Managing your wallet effectively involves several key functions: checking your wallet balance, sending and receiving transactions, and viewing transaction history. This section provides guidelines for each of these functions using the Leaf API interface.

Checking Wallet Balance

To check the balance of your wallet, you will use the following WebSocket commands:

1. **Send Balance Command:**

```
json

{
  "command": "balance",
  "wallet": "[fcc-wallet-address]"
}
```

   - Replace `[fcc-wallet-address]` with your actual wallet address.

2. **Receive Balance Response:**

   - On success:

```
json

{
  "command": "balance",
  "wallet": "[fcc-wallet-address]",
  "balance": [fccamount]
}
```

- On error:

```
json

{
  "command": "balance",
  "error": "[error-message]"
}
```

Sending and Receiving Transactions

To send a transaction, follow these steps:

1. **Initiate NewTransaction:**

```
json

{
  "command": "newtransaction",
  "transid": "[your-transaction-idnr]",
  "pubkey": "[from-wallet-pubkey]",
  "to": [
    {
      "wallet": "[to-wallet-address]",
      "amount": [amount], # Integer of 1/100000000 Coin. 1 = 0.00000001
      "fee": [fee] # Integer of 1/100th of a percent. 1 = 0.01%, 50 = 0.5%
    }
  ]
}
```

- Replace `[your-transaction-idnr]`, `[from-wallet-pubkey]`, `[to-wallet-address]`, `[amount]`, and `[fee]` with your specific details.

2. **Receive NewTransaction Response:**

- On success:

```
json

{
  "command": "newtransaction",
  "transid": "[your-transaction-idnr]",
  "sign": "[transaction-ledger-data-to-sign]",
  "fcctime": "[fcctimestamp]"
}
```

- On error:

```
json

{
  "command": "newtransaction",
  "transid": "[your-transaction-idnr]",
  "error": "[error-message]"
}
```

3. **Sign the NewTransaction:**

```
json
```

```
{
  "command": "signtransaction",
  "transid": "[your-transaction-idnr]",
  "signature": "[your-transaction-ledger-data-signature]"
}
```

- Generate the signature using Ed25519:

```
perl

[your-transaction-ledger-data-signature] = octhex(
  Crypt::Ed25519::sign(
    [transaction-ledger-data-to-sign],
    hexoct([your-wallet-public-key]),
    hexoct([your-wallet-private-key])
  )
);
```

4. **Receive Signature Response:**

- On error:

```
json

{
  "command": "signtransaction",
  "transid": "[your-transaction-idnr]",
  "error": "[error-message]"
}
```

- On success:

```
json

{
  "command": "signtransaction",
  "transid": "[your-transaction-idnr]",
  "transhash": "[node-transaction-id]"
}
```

5. **NewTransaction Processed:**

- On success:

```
json

{
  "command": "processed",
  "transhash": "[node-transaction-id]",
  "wallet": "[wallet-address]",
  "status": "success"
}
```

- On error:

```
json

{
  "command": "processed",
  "transhash": "[node-transaction-id]",
  "error": "[error-message]"
}
```

Viewing Transaction History

The history of transactions can be explored through the explorer API, which is separate from the Leaf API used for wallets and miners.

4.6 – Wallet, Miner Communications and Securities

WebSocket, JSON Communications and Ed25519

All communications for wallets and miners within the FactorialCoin (FCC) network are facilitated through the WebSocket protocol using JSON messages. This approach ensures secure and standardized data exchange between clients (wallets and miners) and network nodes. Additionally, Ed25519 encryption is employed for signing and validating transactions, enhancing the security of interactions with the blockchain.

WebSocket Protocol

WebSocket is utilized to establish full-duplex communication channels over a single TCP connection. This enables efficient and low-latency interaction between clients and nodes, crucial for real-time data exchange in wallet transactions and miner operations.

JSON Messages

JSON (JavaScript Object Notation) is adopted as the format for message exchange between clients and nodes. JSON provides a lightweight, human-readable, and easy-to-parse structure, ensuring compatibility and simplicity in handling data across different platforms and programming languages.

Ed25519 Encryption

Ed25519 serves as the encryption standard for signing and verifying transactions within the FCC network:

- **Signing Transactions:** Wallets use Ed25519 to sign transactions before broadcasting them to nodes. This ensures that transactions are securely authenticated and authorized by the sender.
- **Verifying Transactions:** Nodes validate incoming transactions using the corresponding public keys derived from Ed25519 signatures, ensuring the integrity and authenticity of transactions across the network.

Security Measures

1. **Miner Solution Encryption:** Miners encrypt their solutions with wallet information using a combination of SHA-256 and SHA-512 hashing. This process ensures that only the coinbase server can validate the wallet address associated with the solution, maintaining confidentiality and security during the mining process. By embedding the wallet address securely within the solution data, miners protect their earnings from being tampered with by intermediary nodes. The coinbase server verifies the integrity of the solution by recalculating the hash and comparing it against the transmitted hash, ensuring that the wallet address has not been altered.
2. **Secure Interactions:** Our methods utilize WebSocket for bidirectional communications and JSON for standardization. HTTPS is employed when HTTP API access is needed. Since

open communications eventually end up in the public ledger, we prioritize the identification of sources and the correctness of signatures at the destination. This approach ensures that only the necessary information is used to make transactions possible. Security is maintained privately at both ends, while in-between communications remain open, as they pertain to public data that will be recorded in the public ledger. This balance ensures the confidentiality of sensitive data while leveraging the public nature of the ledger for transparency and integrity.

Summary

FactorialCoin ensures secure, efficient, and reliable interactions between wallets, miners, and network nodes by employing WebSocket for real-time communication and JSON for structured data exchange. Transaction encryption and verification use Ed25519, ensuring that sensitive data is protected while leveraging the public nature of the ledger for transparency and integrity. This combination of technologies and protocols contributes to the robustness and security of the FactorialCoin blockchain ecosystem, supporting secure transactions and decentralized consensus mechanisms.

4.7 - Wallet Contactbook Integration

The integration of a contactbook within the FactorialCoin (FCC) wallet system allows users to manage their interactions and transactions more efficiently. This feature enhances privacy and organization by grouping wallets into various categories such as contacts, public, and private, each serving distinct purposes within the user's financial ecosystem.

JSON Structure

The following JSON structure outlines how the contactbook integrates with the wallet system:

```
json

{
    "encoded": "",
    "wlist": [

// original wallet object
        {
            "name": "WalletName",
            "wallet": "WalletAddress",
            "privkey": "Private Key",
            "pubkey": "Public Key"
        },

// contact wallet group
        {
            "name": "ContactWalletName",
            "contact": {
                "Name": "ContactName",
                "Credit": {
                    "[COIN]": [
                        {
                            "name": "ContactCreditName",
                            "wallet": "CreditWalletAddress"
                        }
                    ]
                },
                "Debit": {
                    "[COIN]": [
                        {
```

```
                            "name": "ContactDebitName",
                            "wallet": "DebitWalletAddress",
                            "privkey": "Private Key",
                            "pubkey": "Public Key"
                        }
                    ]
                }
            }
        },

// public wallet group
        {
            "name": "WalletName",
            "public": {
                "Name": "GroupName",
                "Debit": [
                    {
                        "[COIN]": [
                            {
                                "name": "ContactName",
                                "wallet": "WalletAddress",
                                "privkey": "Private Key",
                                "pubkey": "Public Key"
                            }
                        ]
                    }
                ]
            },
        },

// private wallet group
        {
            "name": "WalletName",
            "private": {
                "Name": "GroupName",
                "Debit": [
                    {
                        "[COIN]": [
                            {
                                "name": "ContactName",
                                "wallet": "WalletAddress",
                                "privkey": "Private Key",
                                "pubkey": "Public Key"
                            }
                        ]
                    }
                ]
            }
        }

    ]
}
```

Explanation

- **encoded:** Placeholder for password encoded data, used for encryption purposes if required.
- **wlist:** Array containing wallet objects, each representing a distinct wallet within the user's portfolio.
    - **name:** Name of the wallet for easy identification.
    - **contact:** Grouping for contact-related wallets.
        - **Name:** Group name for organizational purposes.
        - **Credit:** List of addresses where coins are sent to contacts.
        - **Debit:** List of addresses where coins are received from contacts, including wallet addresses and associated keys.
    - **public:** Grouping for public-facing wallets.
        - **Name:** Group name for organizational purposes.

        - **Debit:** List of addresses for public transactions, including wallet addresses and associated keys.
    - **private:** Grouping for private wallets.
        - **Name:** Group name for organizational purposes.
        - **Debit:** List of addresses for private transactions, including wallet addresses and associated keys.

Additional Notes

- **Credit Wallet List:** This list contains addresses to which coins are sent, typically belonging to contacts. Users cannot control these addresses directly.
- **Debit Wallet List:** This list contains addresses from which coins are received, and from which users can initiate transactions by using their private keys.
- **Combination of Credit and Debit:** Within a contact object, a Debit address serves as a Credit address for the user's contact, enabling transactions to and from this address while maintaining a record of payments made and received.

Optional Information Fields

Additionally, each contact, public, or private object can include optional information fields to provide more personal clarity and facilitate interactions:

- **Name**: Name of the contact or entity.
- **Nick**: Nickname or alias.
- **SurName**: Surname or last name.
- **Email**: Email address.
- **Phone1**: Primary phone number.
- **Phone2**: Secondary phone number.
- **Address**: Physical address.
- **City**: City name.
- **County**: County or region.
- **State**: State or province.
- **Country**: Country name.
- **Postal**: Postal or ZIP code.
- **Continent**: Continent name.
- **Group**: Group affiliation or category.
- **Avatar**: Avatar image for visual identification.

These fields are optional and can be shared by the user as needed for payment requests and other interactions, enhancing the usability and personalization of the wallet system.

Import Considerations

For import purposes, this structured information ensures compatibility with other wallet systems. It facilitates the seamless transfer of wallet data across different platforms, maintaining integrity and functionality. This export-ready format allows for easy integration into new wallet versions, supporting broad usability and interoperability within the FCC ecosystem.

5 - Balance and Transaction Implementation

5.1 – Wallet Balance and Transaction API Overview

FactorialCoin (FCC) implements a comprehensive API for managing wallet balances and transactions, leveraging WebSocket communication and JSON protocols. The API utilizes Ed25519 encryption for secure transactions using a wallets public and private keys.

5.2 – Setting Up a WS connection as Wallet Leaf

To establish a WebSocket (WS) connection as a wallet leaf, follow these steps:

1. **WebSocket Support:** All communication with the FactorialCoin network occurs over WebSocket, except for node list collection, which uses HTTPS.
2. **JSON Communication:** Ensure all communication is in JSON format to interact with the FactorialCoin network effectively.
3. **Ed25519 Encryption:** Use the Ed25519 sign function to sign transactions securely with your wallet's public and private keys.

5.3 – Requesting Wallet Balance

To request the balance of a wallet:

- **Outgoing Command:** Send a JSON request to the node with the command "balance" and specify the wallet address.

```
json

{
  "command": "balance",
  "wallet": "[fcc-wallet-address]"
}
```

- **Incoming Response on Success:**

```
json

{
  "command": "balance",
  "wallet": "[fcc-wallet-address]",
  "balance": "[fcc-amount]"
}
```

- **Incoming Response on Error:**

```
json

{
  "command": "balance",
  "error": "[error-message]"
}
```

5.4 – Setting Up, Signing, and Processed Wallet Transactions

The process for setting up, signing, and processing wallet transactions involves the following steps:

1. **Outgoing Command to Initiate Transaction:**

```
json

{
  "command": "newtransaction",
  "transid": "[your-transaction-idnr]",
  "pubkey": "[wallet-pubkey]",
  "to": [
    {
      "wallet": "[recipient-wallet-address]",
      "amount": "[transaction-amount]",
      "fee": "[transaction-fee]"
    }
  ]
}
```

2. **Incoming Response on Success (After New Transaction Initiation):**

```
json

{
  "command": "newtransaction",
  "transid": "[your-transaction-idnr]",
  "sign": "[transaction-ledger-data-to-sign]",
  "fcctime": "[fcctimestamp]"
}
```

3. **Outgoing Command to Sign Transaction:**

```
json

{
  "command": "signtransaction",
  "transid": "[your-transaction-idnr]",
  "signature": "[your-transaction-ledger-data-signature]"
}
```

4. **Incoming Response on Success (After Transaction Signing):**

```
json

{
  "command": "signtransaction",
  "transid": "[your-transaction-idnr]",
  "transhash": "[node-transaction-id]"
}
```

5. **Incoming Response on Transaction Processing:**

```
json

{
  "command": "processed",
  "transhash": "[node-transaction-id]",
  "wallet": "[wallet-address]",
  "status": "success"
}
```

   - **On Error:**

     ```
     json
     ```

```
{
  "command": "processed",
  "transhash": "[node-transaction-id]",
  "error": "[error-message]"
}
```

5.5 – Transaction Setup Process Fallbacks

Transaction setup processes include fallback mechanisms to handle block limitations effectively. These mechanisms ensure transactions are processed even under adverse conditions such as network delays or congestion.

Handling Large Spendable Amounts

In some cases, a wallet has more coins spendable than the allowed number of inblocks in a single transaction. This is due to the limitation on the number of inblocks that can be used within a single transaction's seal. The current limit is 255 inblocks, which is an integer length of one byte. This scenario commonly occurs with nodes that receive a lot of coinbase fee payouts from miners, where only 127 coins can be spent due to the inblock limit.

Fallback Mechanism

When a transaction setup fails because the spendable amount exceeds the allowed limit, the node provides the maximum spendable amount for a single transaction, including the transaction fee. The user can then calculate the remaining amount to be spent in multiple transaction rounds, ensuring the total intended amount is eventually spent as per the wallet's balance.

Here’s the fallback mechanism in detail:

1. **Failed Transaction Setup Response:** If a transaction setup fails due to too many funds, the node responds with an error message and the maximum spendable amount:

   ```
   json

   {
     "command": "newtransaction",
     "transid": "[your-transaction-idnr]",
     "error": "Too many funds needed to make this transaction. Please split up into smaller
   amounts",
     "spendable": "[maxamount]"
   }
   ```

2. **Recalculate Spendable Amount:** The user recalculates the spendable amount by adjusting for the transaction fee and ensuring it fits within the inblock limit:

   ```
   perl

   $transferAmount = int(($spendable - ($spendable * ($fee / 100))) * 0.95);
   ```

3. **Repeat Transaction Rounds:** The process repeats in several transaction rounds. Each round starts with the total amount left to spend, subtracts the spendable amount, and tries the remaining amount again. This loop continues until the entire amount is spent.

   - **Initial Calculation:** Start with the total amount to spend.

```
perl

$totalAmount = [total-amount-to-spend];
$remainingAmount = $totalAmount;
```

- **Transaction Loop:** Loop through transactions until the remaining amount is zero.

```
perl

while ($remainingAmount > 0) {
  $transferAmount = int(( $spendable - ($spendable * ($fee / 100)) ) * 0.95);
  # Make transaction with $transferAmount
  $remainingAmount -= $transferAmount;
}
```

4. **Example Scenario:** Suppose a node has 500 coins spendable, but due to the inblock limit, only 127 coins can be spent per transaction. The node will return the spendable amount, and the user will split the transaction as follows:
    - Total amount to spend: 500 coins
    - Maximum spendable per transaction: 127 coins
    - Calculate the transaction fee and adjust the spendable amount.
    - Create transactions until the total amount is spent:
        - Transaction 1: Spend 127 coins.
        - Transaction 2: Spend 127 coins.
        - Transaction 3: Spend 127 coins.
        - Transaction 4: Spend 119 coins (remaining amount).

By following this fallback mechanism, users can ensure their transactions are processed even with the inblock limitations.

Future Updates

Future updates will make the fallback process itself obsolete by implementing an extended inblock version. This new design will allow transactions to spend the whole amount to all destination wallets in a single transaction, regardless of the number of inblocks. When a transaction setup fails due to too many funds, the node will switch to using the extended inblock method automatically, handling the entire amount in a single transaction. This will eliminate the need for users to manually split their transactions.

Node's Perl Code for Error Handling:

When the node detects too many funds needed for a single transaction, it responds with an error and the maximum spendable amount:

```
perl

outjson($client, {
  command => 'newtransaction',
  transid => $job->{transid},
  error => "Too many funds needed to make this transaction. Please split up into smaller amounts",
  spendable => $maxamount
});
```

This ensures that users are informed about the transaction limits and can adjust their transaction amounts accordingly.

5.6 – Wallet Event Triggers

Wallet balance updates and transaction confirmations trigger events are crucial for managing history update timers. Given that the explorer has a maximum delay of one minute before processed transactions appear in the explorer databases, best practices dictate minimizing the use of the Explorer API. This section outlines the recommended approach to handle these updates effectively.

Balance Updates and Transaction Confirmations

1. **Initial Balance Check:**
    - When a balance call is made, the application can determine if the balance has changed by checking the last active transaction ID.
    - If the balance has changed, it indicates that a new transaction has occurred.
2. **Triggering History Updates:**
    - Upon detecting a balance change, the application should trigger a timeout of one minute before performing a history update for the wallet.
    - This delay ensures that the explorer database has processed and recorded the new transaction.
3. **Startup Behavior:**
    - On the startup of the wallet interface, a buffer should be built containing the history data to be displayed in the frontend.
    - This buffer allows for quick access to historical data without repeatedly querying the Explorer API.
4. **Updating History:**
    - The history should only be updated when the balance has changed or a transaction has been processed.
    - This approach reduces the frequency of API calls and optimizes performance.
5. **Mining Payouts and Fee Distributions:**
    - Balance changes may also occur due to mining payouts of coinbases or fee distributions.
    - The balance call output will include the type of the new and last seal of the wallet’s spendable blocks, indicating these changes.
6. **Explorer API Usage:**
    - The explorer API is designed for browsing transaction pages, coinbases, and fee payout records.
    - However, the goal is to make the system event-driven, where updates are triggered based on user actions such as changing or starting wallets, balance changes, and browsing transaction pages.

Event Triggers and Timing

1. **Event-Driven Updates:**
    - Events should trigger updates when:
        - Wallets are changed or started.
        - The balance has changed.
        - Pages of transactions are browsed.
2. **Timer Event:**
    - The timer event serves to delay the triggers for updates from the explorer, accommodating the processing delay of transactions from the node's ledger into the explorer database.
    - This helps in avoiding unnecessary API calls and managing the update load efficiently.

Example Implementation

Here’s a step-by-step example of how to handle wallet event triggers:

1. **On Wallet Startup:**
    - Fetch the initial balance and history.
    - Build a buffer with the historical data.

```
javascript

let walletHistory = fetchWalletHistory(walletAddress);
let walletBalance = fetchWalletBalance(walletAddress);
```

2. **Balance Change Detection:**
    - Periodically check the balance.
    - If the balance has changed (last transaction ID is different), trigger a timer for history update.

```
javascript

setInterval(() => {
  let newBalance = fetchWalletBalance(walletAddress);
  if (newBalance.tid !== walletBalance.tid) {
   setTimeout(updateWalletHistory, 60000); // 1-minute delay
  }
  walletBalance = newBalance;
}, 30000); // Check every 30 seconds
```

3. **Update Wallet History:**
    - After the delay, update the wallet’s history.

```
javascript

function updateWalletHistory() {
  walletHistory = fetchWalletHistory(walletAddress);
  displayWalletHistory(walletHistory);
}
```

4. **Handling Mining Payouts and Fees:**

- Include checks for mining payouts and fee distributions in the balance update logic.

```
javascript

function fetchWalletBalance(walletAddress) {
  // Fetch balance and check for mining payouts and fee distributions
  let balanceData = leafApi.Balance(`${walletAddress}`);
  return balanceData;
}
```

By following these guidelines, wallet applications can efficiently manage balance updates and transaction confirmations, ensuring that users have accurate and up-to-date information with minimal API calls. This approach leverages event-driven triggers and timed delays to optimize performance and resource usage.

6 - History and Explorer API Integration

This chapter outlines the integration of transaction history and explorer API for FactorialCoin (FCC) and PTTP. It covers available explorers, API commands, and methods to retrieve various types of transaction history and statistical data.

6.1 - Available Explorers

Below are the URLs for the Explorer API for each symbol:

- **FCC Explorer**: https://factorialcoin.nl:5152/api?command=help
- **PTTP Explorer**: https://factorialcoin.nl:9612/api?command=help

6.2 – Available Explorer API Commands

Here is a detailed explanation of the available commands for the Explorer API.

1. **Request Explorer API Help**
    - `command=help`
    - Returns a JSON-structured basic help output.
2. **Request Statistic Coin Ledger Information**
    - `command=info`
    - This functionality is not yet implemented.
3. **Request Volume Graph Data**
    - `command=graph&volume`
    - Returns a JSON-structured volume graph data.
4. **Request Activity Graph Data**
    - `command=graph&activity`
    - Returns a JSON-structured activity graph data.
5. **Request Seal by ID**
    - `command=get&seal=[sealid]`
    - Returns a transaction seal by Seal ID.
6. **Request Transaction by ID**
    - `command=get&tid=[transid]`
    - Returns a transaction seal by Transaction ID.
7. **Request Transaction, Coinbase, Fee, or Raw History for Wallet Address**
    - `command=get&wallet=[address]`
    - **Optional Filters**:
        - `&filter=[balance||coinbase||fee]` (Omission returns all types in raw seal output)

- `&pagesize=[1..items]` (Omission gives full record list, this will be large downloads for very old, nodes and mining wallets to fully download, advisable is a 1000 items max per page to keep interactive flow high, for if you fully download and process the history of these wallets)
- `&start=[fcctime]` (Omission defaults to unused and 0)
- `&startpage=[0..pages]` (Omission defaults to 0)
- `&startpos=[nextpos]` (Omission defaults to unused and 0)
- `&startseal=[sealid]` (Omission defaults to unused and 0)
- `&descending=[0||1]` (Omission defaults to 0)

6.3 – Statistic Coin Ledger Information

This functionality is currently not implemented.

6.4 – Transaction History

To retrieve transaction history from the Explorer, use the following command:

`https://factorialcoin.nl:5152/api?command=get&wallet=[address]&filter=balance`

- **Filter by Type**:
  - `&filter=[balance]` - To get balance transactions.
  - `&filter=[coinbase]` - To get coinbase transactions.
  - `&filter=[fee]` - To get fee transactions.
  - Omission returns all types in raw seal output.
- **Limit Output**:
  - `&pagesize=[1..items]` - Limits the number of items per page.
  - `&startpage=[0..pages]` - Start listing from a specific page.
  - `&start=[fcctime]` - Start listing from a specific FCC time.
  - `&startpos=[nextpos]` - Start listing from the next ledger block position.
  - `&startseal=[sealid]` - Start listing from a specific seal ID.
  - `&descending=[0||1]` - Order by FCC time (0 = start to end, 1 = end to start).

6.5 – Coinbase History

To retrieve coinbase history, use the following command with the filter set to `coinbase`:

`https://factorialcoin.nl:5152/api?command=get&wallet=[address]&filter=coinbase`

Follow the same optional parameters for pagination and filtering as outlined in section 6.4.

6.6 – Fee History

To retrieve fee payout history, use the following command with the filter set to `fee`:

```
https://factorialcoin.nl:5152/api?command=get&wallet=[address]&filter=fee
```

Follow the same optional parameters for pagination and filtering as outlined in section 6.4.

6.7 – RAW Ledger History

To retrieve raw ledger history, which includes wallet, coinbase, and fee transactions, use the command without specifying a filter:

```
https://factorialcoin.nl:5152/api?command=get&wallet=[address]
```

Follow the same optional parameters for pagination and filtering as outlined in section 6.4.

6.8 – Transaction by Seal-ID

To retrieve a transaction seal by Seal ID, use the following command:

```
https://factorialcoin.nl:5152/api?command=get&seal=[sealid]
```

This will return the transaction seal associated with the specified Seal ID.

6.9 – Transaction by Transaction-ID

To retrieve a transaction seal by Transaction ID, use the following command:

```
https://factorialcoin.nl:5152/api?command=get&tid=[transid]
```

This will return the transaction seal associated with the specified Transaction ID.

6.10 – Volume Release & Transaction Activity Graph Data

To retrieve released volume and transaction activity graph data, use the following commands:

1. **Volume Graph Data**
    - `command=graph&volume`
    - Returns a JSON-structured volume graph data.
2. **Activity Graph Data**
    - `command=graph&activity`
    - Returns a JSON-structured activity graph data.

7. Mining API Implementation

In this chapter, we'll delve into the detailed implementation of the Mining API for FactorialCoin (FCC), using the provided Perl modules and script (`FCC::miner` and `fccminer.cgi`). This will provide a comprehensive understanding of how miners can participate in the network to mint new coins.

7.1 – Introduction to Mining in FactorialCoin

FactorialCoin employs a Proof-of-Work (PoW) mechanism where miners solve cryptographic challenges to mint new coins. Unlike traditional blockchain networks, FactorialCoin miners focus solely on generating valid solutions to earn new coins for the coinbase. The network's nodes independently validate transactions, while miners contribute computational resources to solve factorial permutation challenges.

The mining process in FactorialCoin revolves around calculating factorial permutations efficiently. Miners aim to find a specific permutation that meets the network's predefined difficulty criteria. This computational effort ensures the security and decentralization of the FactorialCoin network by adding new coins into circulation through a competitive and decentralized mining process.

7.2 – Setting Up the Mining Environment

Before miners can start minting coins, they need to set up their mining environment, configure connections to the FactorialCoin node, and prepare their wallets.

Configuring Node and Wallet

```
perl

my $NODEIP = '141.138.137.123';  # Example IP address of the FactorialCoin node
my $NODEPORT = 7050;             # Example port number for the node

# Wallet address where mined coins will be deposited
my $WALLET = 'YOUR_FCC_WALLET_ADDRESS';
```

Node Selection

Miners can either specify a node or select a random node from a list retrieved from a centralized service.

```
perl

# Choosing a specific node or opting for a random node
if (-e 'minerrandom.fcc') {
    # Fetch random node list from a centralized service
    my $nodes = gclient::website('https://factorialcoin.nl:5151/?nodelist');
    if ($nodes->content()) {
        my @nl = split(/\s/, $nodes->content());
        my $node = $nl[int(rand(1 + $#nl))];
        ($NODEIP, $NODEPORT) = split(/:/, $node);
    }
}
```

7.3 – Multithreaded Mining Setup

The mining process is optimized using multithreading (`gthreads`) to parallelize the computation of factorial permutations across multiple threads.

Thread Configuration

```
perl
```

```
# Number of threads for mining
my $THREADS = 3;  # Example: 3 threads

# If threads configuration file exists, read it; otherwise, prompt for input
if (!-e 'threads_mining.fcc') {
    print "On how many threads should I mine? (3) > ";
    $THREADS = <STDIN>;
    chomp $THREADS;
    if (!$THREADS) { $THREADS = 3; }
    elsif ($THREADS =~ /[^0-9]/) {
        print "> I don't understand!\n";
        exit;
    }
} else {
    $THREADS = gfio::content("threads_mining.fcc");
}
```

7.4 – Mining Algorithm and Functions

FactorialCoin mining involves calculating permutations of factorial numbers and hashing solutions to find valid blocks.

Factorial Calculation

```
perl

sub fac {
    my ($f) = @_;
    my $fac = 1;
    while ($f > 1) {
        $fac *= $f;
        $f--;
    }
    return $fac;
}
```

Initialization of Permutations

```
perl

sub initperm {
    my ($len) = @_;
    my $p = "";
    for my $i (0 .. $len - 1) {
        $p .= chr(65 + $i);  # Generates permutations like 'A', 'B', 'C', ..., 'Z'
    }
    return $p;
}
```

Permutation Generation

```
perl

sub perm {
    my ($init, $k) = @_;
    my $n = length($init);
    my $dn = $n;
    my $out = "";
    my $m = $k;
    for (my $i = 0; $i < $n; $i++) {
        my $ind = $m % $dn;
        $out .= substr($init, $ind, 1);
        $m = $m / $dn;
        $dn--;
        substr($init, $ind, 1, substr($init, $dn, 1));
```

```
    }
    return $out;
}
```

7.5 – Mining Loop and Solution Submission

The main mining loop iteratively retrieves mining tasks from the node, computes factorial permutations, and submits solutions.

Main Mining Loop (`fccminer.cgi`)

```
perl

while ($leaf && !$leaf->{quit}) {
    mineloop();
    $leaf->leafloop();
}
```

Mining Function (`mineloop`)

let's break down the mining function `mineloop` and explain how the miner threads are managed within it:

```
perl

sub mineloop {
    if (!$PROBLEM) {
        usleep(10000);  # Wait for new problem assignment
        return;
    }

    # Multithreaded mining setup
    if (gthreads::running() < $THREADS) {
        # Spawning threads for concurrent mining
        $LOOP++;
        print "Starting thread $LOOP/$THREADS ...\n";
        my $problem = $PROBLEM;
        my $pos = $POS;
        my $fac = $FAC;
        my $psize = $PSIZE;
        my $hint = $HINT;
        my $ehints = $EHINTS;

        # Start a new thread for mining
        if ($LOOP == $THREADS) {
            gthreads::start('miner', \&minerthread, $problem, $pos, $fac, $psize + $PREST, $hint,
$ehints);
            $LOOP = 0;
            $POS += $PSIZE + $PREST;
            if ($HINT) {
                $HINTPOS++;
                $HINT = substr($HINTS, $HINTPOS, 1);
            }
        } else {
            gthreads::start('miner', \&minerthread, $problem, $pos, $fac, $psize, $hint, $ehints);
            $POS += $PSIZE;
        }

        if ($POS > $FAC) {
            $POS -= $FAC;
        }
    }

    # Monitoring hash rate and displaying mining statistics
    if (time - $DISPTIME > 60) {
        my $hr = hashrate();
        my $p = gthreads::read('miner', 'problemcount');
```

```
        my $s = gthreads::read('miner', 'success');
        my $dp = $p - 1;
        my $perc = '0';
        if ($dp) {
            $perc = int(10000 * $s / $dp) / 100;
        }
        my $str = "$hr Fhs - Problems $p - Found solutions $s ($perc %) - Threads $THREADS";
        my $th = gthreads::read('miner', 'tothash');
        my $dh = $th - $HASHSTART;
        my $hp = int(10000 * $dh / $PROBLEM->{diff}) / 100;
        my $dtm = int((time - $RUNTIME) / 60);
        my $hour = int($dtm / 60);
        my $min = sprintf("%02d", $dtm % 69);
        my $days = int($hour / 24);
        $hour = sprintf("%02d", $hour % 24);
        my $str2 = "$days" . 'd' . ":$hour" . 'h' . $min . "m Curr: $hp% Diff=$PROBLEM->{diff}
Len=$PROBLEM->{length} Hints=$PROBLEM->{hints} ($PROBLEM->{ehints})";
        my $sl = length($str);
        my $sp = (90 - $sl) >> 2;
        my $spl = "";
        if ($sl % 2) {
            $spl = ' ';
        }
        print '   ';
        print '-' x 73;
        print "\n";
        print ' ' x $sp;
        print "$str\n";
        print ' ' x $sp;
        print "$str2\n";
        print '   ';
        print '-' x 73;
        print "\n";
        $DISPTIME = time;
    }

    usleep(100000);
}
```

Explanation:

1. **Initial Check (`if (!$PROBLEM)`):**
    - This condition checks if there is a current mining problem (`$PROBLEM`). If there isn't, it sleeps for a short duration (`usleep(10000)`) and then returns, waiting for a new problem assignment from the FactorialCoin node.
2. **Multithreaded Mining Setup (`if (gthreads::running() < $THREADS)`):**
    - Here, the function checks how many threads (`gthreads::running()`) are currently active compared to the total desired threads (`$THREADS`). If there are fewer threads running than specified, it proceeds to spawn new threads for concurrent mining.
    - `$LOOP++` increments the loop counter to keep track of the number of times this block has executed.
3. **Starting Miner Threads:**
    - The actual threads are started using `gthreads::start`. It passes the following parameters to `minerthread`:
        - `$problem`: Current mining problem details.
        - `$pos`: Current position in the permutation sequence.
        - `$fac`: Total factorial permutations.

        - `$psize`: Size of the permutation block each thread should handle.
        - `$hint`: Primary hint for the permutation.
        - `$ehints`: Additional hints for more complex permutations.
    - Depending on the value of `$LOOP`, the function adjusts the starting position (`$POS`) and increments the hint position (`$HINTPOS`) if hints are being used.

4. **Monitoring and Displaying Mining Statistics:**
    - Every 60 seconds (`if (time - $DISPTIME > 60)`), the function calculates and displays various mining statistics:
        - **Hash rate (`$hr`)**: Calculated based on total hashes processed.
        - **Problem count (`$p`)**: Number of problems encountered.
        - **Success count (`$s`)**: Number of successful solutions found.
        - **Success rate (`$perc`)**: Percentage of successful solutions compared to total problems.
        - **Hash power (`$hp`)**: Percentage of difficulty solved.
        - **Elapsed time (`$dtm`)**: Time elapsed since mining started.
        - **Details (`$str2`)**: Additional details including difficulty, length, and hints used.
    - These statistics provide real-time feedback on the mining operation's progress and efficiency.
5. **Usleep (`usleep(100000)`):**
    - Finally, the function sleeps for 100,000 microseconds (0.1 seconds) before starting the next iteration of the loop. This prevents excessive CPU usage by throttling the mining loop.

This `mineloop` function effectively manages the multithreaded mining process, ensures efficient utilization of computational resources, and provides comprehensive statistics for monitoring and optimizing mining performance in FactorialCoin. Adjustments to the thread count (`$THREADS`) and mining parameters can be made based on network conditions and hardware capabilities to maximize mining efficiency.

Mining Thread (`minerthread`)

Let's break down the `minerthread` subroutine and explain its functionality:

```
perl

sub minerthread {
    my ($id, $problem, $pos, $fac, $psize, $hint, $ehints) = @_;

    print "Thread $id: pos=$pos, fac=$fac, psize=$psize, hint=$hint, ehints=$ehints\n";

    my $init = "";
    my $ehint = "";

    # Handling additional hints if present
    if ($ehints) {
        for (my $ehp = 0; $ehp < length($ehints); $ehp++) {
            $ehint = substr($ehints, $ehp, 1);

            # Skip the hint if it matches the primary hint
```

```
            if ($ehint eq $hint) {
                next;
            }

            $init = "";

            # Generating initialization string excluding primary and additional hints
            for (my $i = 0; $i < $problem->{length}; $i++) {
                my $c = chr(65 + $i);  # Generates characters 'A' to 'Z'
                if (($c ne $hint) && ($c ne $ehint)) {
                    $init .= $c;
                }
            }

            # Determine the end position of the block to mine
            my $end = $pos + $psize - 1;
            if ($end > $fac) {
                my $rst = $end - $fac;
                $end = "$fac..$rst";
            }

            # Print thread-specific mining range
            print " -> $id $hint$ehint [$pos..$end]\n";

            # Call minerblock function to perform actual mining
            if (minerblock($id, $init, $problem->{challenge}, $problem->{coincount}, $pos, $fac,
$psize, $hint, $ehint)) {
                gthreads::done($id);  # Mark thread as done and return
                return;
            }
        }
    } else {
        # If no additional hints, generate initialization string excluding primary hint only
        for (my $i = 0; $i < $problem->{length}; $i++) {
            my $c = chr(65 + $i);
            if ($c ne $hint) {
                $init .= $c;
            }
        }

        # Call minerblock function to perform actual mining
        minerblock($id, $init, $problem->{challenge}, $problem->{coincount}, $pos, $fac, $psize,
$hint, "");
    }

    # Mark thread as done
    gthreads::done($id);
}
```

Explanation:

1. **Thread Initialization (`my ($id, $problem, $pos, $fac, $psize, $hint, $ehints) = @_;`):**
    - This subroutine is called for each miner thread, receiving parameters:
        - `$id`: Thread identifier.
        - `$problem`: Current mining problem details.
        - `$pos`: Starting position in the permutation sequence.
        - `$fac`: Total factorial permutations.
        - `$psize`: Size of the permutation block each thread should handle.
        - `$hint`: Primary hint character for the lighter permutation.
        - `$ehints`: Additional hints (optional) for more lighter permutations.
2. **Printing Thread Information (`print "Thread $id: pos=$pos, fac=$fac, psize=$psize, hint=$hint, ehints=$ehints\n";`):**

- This line prints out the thread's identifier (`$id`) and the parameters associated with it, providing visibility into the thread's current state and task.

3. **Handling Additional Hints (`if ($ehints)`):**
    - If additional hints (`$ehints`) are provided:
        - The subroutine iterates over each character in `$ehints`.
        - For each character (`$ehint`), it generates an initialization string (`$init`) excluding both the primary hint (`$hint`) and the current additional hint (`$ehint`).
        - It calculates the end position of the block to mine (`$end`), adjusting if it exceeds `$fac`.
        - It prints out a message detailing the thread's range of work (`print " -> $id $hint$ehint [$pos..$end]\n";`).
4. **Generating Initialization String (`$init`):**
    - The initialization string (`$init`) is generated to exclude specific characters (`$hint` and `$ehint`), ensuring that permutations generated do not contain these characters.
5. **Mining Operation (`minerblock` Function):**
    - The `minerblock` function is called to perform the actual mining operation:
        - It passes parameters such as thread identifier (`$id`), initialization string (`$init`), challenge details (`$problem->{challenge}`), coin count (`$problem->{coincount}`), position (`$pos`), factorial (`$fac`), permutation size (`$psize`), primary hint (`$hint`), and additional hint (`$ehint`).
        - If `minerblock` returns true (indicating success in finding a solution), the thread is marked as done (`gthreads::done($id);`) and the subroutine returns.
6. **No Additional Hints (`else` block):**
    - If no additional hints are provided (`$ehints` is false or empty), the subroutine generates the initialization string (`$init`) excluding only the primary hint (`$hint`).
    - It then calls `minerblock` with an empty string (`""`) for the additional hint parameter.
7. **Thread Completion (`gthreads::done($id);`):**
    - After completing its task (whether finding a solution or exhausting all possibilities), the thread marks itself as done using `gthreads::done($id);`.

This subroutine (`minerthread`) encapsulates the logic for each individual mining thread, managing initialization, permutation generation, and actual mining operations based on the provided hints and problem parameters. It operates in conjunction with the `mineloop` function to achieve efficient and parallelized mining in the FactorialCoin system.

Mining a Block (`minerblock`)

The `minerblock` function is a core component in the mining process, responsible for generating and testing permutations to find a solution that matches a given challenge. Here's a breakdown of the function, line by line, explaining its purpose and logic.

`minerblock` Function Code

```
perl

sub minerblock {
  my ($id, $init, $challenge, $coincount, $pos, $fac, $psize, $hint, $ehint) = @_;
  my $stm = gettimeofday(); my $mc = 0; my $cnt = 0; my $try; my $perm; my $m; my $dn; my $ind; my
$thint = $hint . $ehint; my $ilen = length($init);
  while ($psize > 0) {
    $cnt = 0; my $todo = 50000;
    if ($todo > $psize) { $todo = $psize }
    $psize -= $todo;
    for (my $i = 0; $i < $todo; $i++) {
      $cnt++;
      $try = $init; $perm = ""; $m = $pos; $dn = $ilen;
      while ($dn > 0) {
        $ind = $m % $dn;
        $m = $m / $dn;
        $dn--;
        $perm .= substr($try, $ind, 1, substr($try, $dn, 1));
      }
      #$perm=perm($init,$pos);
      my $hash = minehash($coincount, $thint . $perm);
      if ($hash eq $challenge) {
        gthreads::write('miner', 'solution', $thint . $perm);
        addhash($cnt);
        return 1;
      }
      $pos++; if ($pos >= $fac) { $pos = 0 }
    }
    $mc++; addhash($todo);
    # new challenge signalled?
    if (gthreads::read('miner', 'new')) {
      return 1;
    }
    if ($mc == 10) {
      $mc = 0;
      my $tm = gettimeofday();
      my $dtm = $tm - $stm;
      $stm = $tm;
      my $hr = int(500000 / $dtm);
      print " [ HR $id = $hint : $hr Fhs ]\n";
    }
  }
  return 0;
}
```

Function Breakdown

1. **Function Signature and Variable Initialization:**

   ```
   perl

   sub minerblock {
     my ($id, $init, $challenge, $coincount, $pos, $fac, $psize, $hint, $ehint) = @_;
     my $stm = gettimeofday(); my $mc = 0; my $cnt = 0; my $try; my $perm; my $m; my $dn;
     my $ind; my $thint = $hint . $ehint; my $ilen = length($init);
   ```

   - **Parameters:**
     - `$id`: Thread ID.
     - `$init`: Initial string to be permuted.

        - `$challenge`: Hash challenge to solve.
        - `$coincount`: Number of coins.
        - `$pos`: Current position in permutation.
        - `$fac`: Factorial limit.
        - `$psize`: Permutation size.
        - `$hint`, `$ehint`: Hints used in permutations.
    - **Local Variables:**
        - `$stm`: Start time for performance measurement.
        - `$mc`: Mine counter for performance tracking.
        - `$cnt`: Counter for processed permutations.
        - `$try`, `$perm`, `$m`, `$dn`, `$ind`: Variables used in permutation generation.
        - `$thint`: Combined hints.
        - `$ilen`: Length of the initial string.

2. **Main Loop:**

```
perl

while ($psize > 0) {
  $cnt = 0; my $todo = 50000;
  if ($todo > $psize) { $todo = $psize }
  $psize -= $todo;
```

    - The loop continues until all permutations are exhausted.
    - `50000` permutations are processed in each batch or less if `$psize` is smaller.

3. **Permutation and Hash Generation:**

```
perl

for (my $i = 0; $i < $todo; $i++) {
  $cnt++;
  $try = $init; $perm = ""; $m = $pos; $dn = $ilen;
  while ($dn > 0) {
    $ind = $m % $dn;
    $m = $m / $dn;
    $dn--;
    $perm .= substr($try, $ind, 1, substr($try, $dn, 1));
  }
  my $hash = minehash($coincount, $thint . $perm);
  if ($hash eq $challenge) {
    gthreads::write('miner', 'solution', $thint . $perm);
    addhash($cnt);
    return 1;
  }
  $pos++; if ($pos >= $fac) { $pos = 0 }
```

    - For each permutation:
        - Initialize variables and generate a new permutation (`$perm`).
        - Compute the hash using `minehash` with the combined hints and the permutation.
        - If the generated hash matches the challenge, the solution is written using `gthreads::write` and the function returns `1` (success).
        - Update the position and reset if it reaches the factorial limit.

4. **Performance Tracking and New Challenge Check:**

```
perl
```

```
$mc++; addhash($todo);
if (gthreads::read('miner', 'new')) {
  return 1;
}
if ($mc == 10) {
  $mc = 0;
  my $tm = gettimeofday();
  my $dtm = $tm - $stm;
  $stm = $tm;
  my $hr = int(500000 / $dtm);
  print " [ HR $id = $hint : $hr Fhs ]\n";
}
```

- Update the mine counter and check for a new challenge.
- Every 500,000 hashes (`mc == 10`), calculate and print the hash rate (Fhs).

5. **Function End:**

```
perl

return 0;
```

- If no solution is found after all permutations, the function returns `0` (failure).

By understanding this detailed breakdown, miners can grasp the intricacies of the `minerblock` function, enabling them to optimize and ensure efficient mining operations in the FactorialCoin system.

7.6 – Handling Mining Events

Miners handle events such as new problems, successful solutions, and node connectivity using callback functions (`\&handle`).

Event Handling (`handle` Function)

```
perl

sub handle {
    my ($leaf, $command, $data) = @_;
    if ($command eq 'mine') {
        # New mining task received from the node
        newproblem($data);
    } elsif ($command eq 'solution') {
        # Handling solution validation response from the node
        if ($data->{error}) {
            print " * The core REJECTED our solution :( *\n";
        } else {
            print " * The core accepted our solution :) *\n";
            gthreads::inc('miner', 'success', 1);
        }
    }
}
```

7.7 – Visualizing the Mining Process in FactorialCoin

To grasp the mining process in FactorialCoin thoroughly, let's visualize how miners attempt to solve cryptographic challenges and mint new coins through factorial permutation calculations. This chapter provides a detailed explanation of the mining process, including the idea behind mining, the difficulty involved, and how miners strive to be the first to find a solution.

Understanding the Mining Concept

Mining in FactorialCoin is based on a Proof-of-Work (PoW) mechanism, where miners compete to find a specific permutation of factorial numbers that satisfies the network's difficulty criteria. Unlike traditional blockchain networks, FactorialCoin miners are solely focused on minting new coins for the coinbase, while transaction validation is handled independently by the network's nodes.

Idea Behind Mining

The core idea of mining in FactorialCoin revolves around generating a valid solution to a factorial permutation challenge. Miners aim to compute permutations of factorial numbers efficiently, using computational power to check each permutation against a cryptographic hash.

Difficulty and Permutation Checking

Factorial permutations grow exponentially in complexity as the factorial number increases. For example, the number of permutations for a factorial sequence of length n can be n! (n factorial), which is $1 \times 2 \times 3 \times \ldots \times n$. This complexity increases the difficulty of finding a valid solution, as miners must iterate through a vast number of possible permutations.

Mining Process Visualized

1. **Generating Factorial Permutations**: Miners begin by generating permutations of factorial sequences based on the length specified by the network. The permutations are created using algorithms that rearrange sequences of characters or numbers.

```
perl

sub initperm {
    my ($len) = @_;
    my $p = "";
    for my $i (0 .. $len - 1) {
        $p .= chr(65 + $i);  # Generates permutations like 'A', 'B', 'C', ..., 'Z'
    }
    return $p;
}

sub perm {
    my ($init, $k) = @_;
    my $n = length($init);
    my $dn = $n;
    my $out = "";
    my $m = $k;
    for (my $i = 0; $i < $n; $i++) {
        my $ind = $m % $dn;
        $out .= substr($init, $ind, 1);
        $m = $m / $dn;
        $dn--;
        substr($init, $ind, 1, substr($init, $dn, 1));
    }
    return $out;
}
```

2. **Hashing and Solution Verification**: Once a permutation is generated, miners append it to other necessary data and hash the combination. The resulting hash is then compared with the challenge provided by the network. If the hash matches the challenge, the miner has found a valid solution.

```
perl

sub minehash {
    my ($coincount, $suggest) = @_;
    return securehash($COIN . dechex($coincount, 8) . $suggest);
}
```

3. **Competing for the Solution**: Miners compete with each other to be the first to find and submit a valid solution. When multiple miners find a solution simultaneously, only the first submitted solution triggers the coinbase reward and initiates a new challenge for miners.
4. **Multithreaded Mining**: To optimize performance, mining in FactorialCoin uses multithreading. Each thread independently computes factorial permutations, enhancing the chances of discovering a solution quickly.

```
perl

sub mineloop {
    if (!$PROBLEM) {
        usleep(10000);  # Wait for new problem assignment
        return;
    }

    # Multithreaded mining setup
    if (gthreads::running() < $THREADS) {
        # Spawning threads for concurrent mining
        $LOOP++;
        print "Starting thread $LOOP/$THREADS ...\n";
        # Code for starting miner threads
    }

    # Monitoring hash rate and displaying mining statistics
    if (time - $DISPTIME > 60) {
        # Display mining statistics (hash rate, success rate, etc.)
    }
    usleep(100000);
}
```

Conclusion

Understanding the mining process in FactorialCoin involves grasping the intricacies of factorial permutations, cryptographic hashing, and competitive validation. By implementing the provided Perl modules and script (FCC::miner and fccminer.cgi), developers can participate in the FactorialCoin network, contributing computational resources to mint new coins efficiently. The integration of multithreading ensures that miners can optimize their mining operations, striving to be the first to solve each challenge and earn coinbase rewards.

This visual representation elucidates the mining process's complexity while emphasizing its simplicity in implementation using Perl and the provided mining modules.

7.8 – Security Functionality

In the FactorialCoin system, ensuring the integrity and authenticity of solutions submitted by miners is crucial to prevent fraud and ensure fair reward distribution. The solution employs a security mechanism where miners encrypt their wallet information with the solution before sending it to the node. This encryption prevents nodes from altering the miner's wallet address since the node requires a valid solution to decrypt and retrieve the wallet address, while the Coinbase-Server and miner know the solution to decrypt and encrypt the wallet and solution with the given wallet.

Explanation of the Security Mechanism

1. `solution($leaf, $WALLET, solhash($WALLET, $sol));`

The `solution` function encapsulates the process where a miner constructs and sends their solution along with their encrypted wallet information (`$WALLET`). Here's how it works:

- **Parameters**:
  - `$leaf`: Represents the miners Node-leaf client.
  - `$WALLET`: Miner's wallet address, used for encrypted solution.
  - `solhash($WALLET, $sol)`: Generates a secure hash of the encrypted wallet address and the solution.

2. `solhash` Function

```
perl

sub solhash {
    my ($wallet, $solution) = @_;
    return securehash($wallet . $solution);
}
```

- **Purpose**: Computes a secure hash of the concatenated string `$wallet . $solution`.
- **Usage**: This hash serves as a unique identifier and proof that the miner's wallet and solution are authentic and unchanged.

3. `securehash` Function

```
perl

sub securehash {
    my ($code) = @_;
    if (!$code) { error "FCC.Global.SecureHash: No Code given to hash!" }
    return uc(sha256_hex(sha512_hex($code)));
}
```

- **Purpose**: Provides a secure hashing mechanism to protect against tampering and ensure data integrity.
- **Steps**:
  - **Inner Hashing**: Applies SHA-512 hashing (`sha512_hex`) to `$code`.
  - **Outer Hashing**: Applies SHA-256 hashing (`sha256_hex`) to the result of the inner hash.
  - **Finalization**: Converts the hash to uppercase (`uc`) for consistency.

Explanation of Security Assurance

Encryption of Wallet

When a miner submits a solution in FactorialCoin, they include their wallet address encrypted with the solution (`$sol`). This encryption ensures that the wallet address is securely embedded within the solution data. This step prevents nodes from altering the miner's wallet address during transmission to the coinbase server.

Hashing for Integrity

The `solhash` function is integral to ensuring the integrity and authenticity of the combined data (encrypted wallet and solution). Here’s how it works:

- **Purpose**: `solhash` computes a hash that incorporates both the encrypted wallet (`$wallet`) and the solution (`$solution`).
- **Process**: It applies a series of cryptographic hashing functions (`sha512_hex` followed by `sha256_hex`) to generate a unique hash value (`$hash`). This hash value serves as a cryptographic proof that the combined data (wallet and solution) has not been tampered with.

Coinbase Verification Process

Once nodes receive solutions from miners, they pass these solutions to the coinbase server for validation. Here's how the validation process ensures security:

- **Step-by-Step Validation**:
    1. **Receive Solution**: Nodes receive solutions from miners, each containing an encrypted wallet address and a solution.
    2. **Compute Hash**: Nodes compute the hash (`solhash`) of the received solution using the provided solution.
    3. **Comparison**: The computed hash (`$hash_computed`) is compared against the transmitted hash (`$hash_transmitted`).
    4. **Match Verification**: If `($hash_computed eq $hash_transmitted)`, it confirms that:
        - The wallet address embedded within the solution has not been altered.
        - The solution is authentic and has not been tampered with en route to the node.
    5. **Coinbase Transaction**: Upon verification, the coinbase server can proceed to create a coinbase transaction for the miner. This transaction includes the miner's wallet address, ensuring that the miner receives the rightful rewards for their contribution.

Importance for Miners

Implementing this security mechanism is crucial for miners because it:

- **Ensures Data Integrity**: By hashing the solution and the encrypted wallet, miners protect against unauthorized modifications.
- **Verifies Authenticity**: Nodes and the coinbase server verify that the submitted solution originates from the correct miner with an unaltered wallet address.
- **Secures Reward Distribution**: Miners can confidently receive their rewards without the risk of fraudulent wallet address changes by nodes.

In conclusion, the combination of wallet encryption and hashing mechanisms in FactorialCoin provides robust security measures. These measures safeguard miners' solutions, ensuring that only legitimate, unaltered submissions are processed for rewards to miners and nodes on the blockchain.

8 – Best Practices

8.1 – Connection Error Handling

Nodelist-Service connection error handling and practices.

In cases where the nodelist service is down for a restart to update to a new version and cannot be reached, your software should always have a backup nodelist. This backup allows you to keep track of online and offline nodes. Nodes will only accept connections if they are part of the online core node network. During this time, nodes that are online can be reached for balance and transaction calls.

Since the nodelist service also intermediates the coinbase release, miners will be down during this period. The coinbase server is designed to handle this by lowering the release difficulty, ensuring that the release catches up for the missed period.

Node Connection Error Handling and Practices

When nodes go offline, they should be marked as offline within your backup nodelist. This list should be recycled and randomly rotated to use the online nodes evenly. This practice helps to spread the network load and maintain consistent performance.

Explorer Connection Error Handling and Practices

Currently, there is only one central Explorer-Service. In the event of an offline error, a local buffer should be used to store the last known records of the wallet's history. Once the Explorer-Service is back online, your wallet app can update this local buffer to display any new records in the frontend.

Future updates will aim to split the wallet API from the central Explorer server, creating a decentralized option for the explorer over WebSocket. This decentralized explorer would function similarly to a node but be dedicated to wallet history services. This approach would decentralize the network load for the explorer API, support potentially millions of wallets, and allow wallets to register for history update push messages. This would eliminate the need for a history timer to pull updates from the current central Explorer API server.

8.2 – Balance Error Handling

Technically, the only reason to receive a balance error response is when you request the balance for a wallet ID that is not listed in the ledger or an invalid wallet address for the symbol.

Handling Unlisted Wallet IDs:

- **Error Identification**: When your application requests the balance of a wallet ID not found in the ledger, an error response will be generated.

- **Error Logging**: Log this error with details about the requested wallet ID for debugging and record-keeping purposes if this scenario is needed.
- **User Notification**: Notify the user that the requested wallet ID is not listed in the ledger. Provide a user-friendly message indicating the issue.

Handling Invalid Wallet Addresses:

- **Error Identification**: If the wallet address provided does not conform to the expected format or symbol, an error response will occur.
- **Error Logging**: Log this error with details about the invalid wallet address for troubleshooting if this scenario is needed.
- **User Notification**: Inform the user that the wallet address is invalid. Offer guidance on the correct format or symbol for wallet addresses to help prevent future errors.

Best Practices:

1. **Input Validation**: Implement validation checks on wallet IDs and addresses before sending balance requests. This preemptively catches errors and reduces unnecessary requests.
2. **Graceful Degradation**: Ensure that your application can gracefully handle balance errors without crashing or becoming unresponsive.
3. **Retry Mechanism**: Consider implementing a retry mechanism for balance requests in case of temporary issues, but avoid excessive retries to prevent overloading the network.
4. **User Experience**: Provide clear and actionable error messages to users, helping them understand and rectify the issue if this scenario is needed.
5. **Logging and Monitoring**: Maintain logs of all balance errors and monitor them regularly to identify patterns or potential issues that may require further attention if this scenario is needed.

8.3 – Transaction Error Handling

Transaction errors can occur even when protocols are followed, due to various factors such as insufficient balance or attempts at double-spending. Below are some scenarios and best practices to handle these errors:

Scenarios Leading to Transaction Errors:

1. **Insufficient Balance**:
    - **Description**: Attempting to spend more funds than available in the wallet.
    - **Error Handling**:
        - **Error Identification**: Detect insufficient balance before creating the transaction.
        - **Error Logging**: Log the attempted transaction details.
        - **User Notification**: Inform the user about the insufficient balance.
2. **Double-Spending**:
    - **Description**: Creating multiple transactions with the same wallet on different nodes and sending them simultaneously.

- **Error Handling**:
    - **Error Identification**: Nodes initially validate each transaction independently, but only the first transaction processed into the ledger will succeed; the rest will fail due to attempting to spend the same inblocks.
    - **Error Logging**: Log the failed transactions.
    - **User Notification**: Notify the user that only the first transaction was successful, and subsequent attempts failed due to double-spending.

Best Practices for Preventing Transaction Errors:

1. **Node Consistency for Sequential Actions**:
    - Ensure all transactions from the same wallet are processed on the same node until the sequence of actions is completed. This prevents the risk of conflicting transactions being validated by different nodes.
2. **Optimizing Network Load with Threaded Transactions**:
    - For actions involving different wallets, distribute the transactions across different nodes to balance the network load and improve performance.
3. **Transaction Validation**:
    - **Pre-Send Validation**: Validate transactions locally for sufficient balance and uniqueness of inblocks before sending them to the network.
    - **Post-Send Validation**: Confirm the status of each transaction after sending and handle any errors by notifying the user and logging the details.

Handling Protocol Errors:

- **Error Identification**: Identify errors resulting from protocol deviations, such as incorrect transaction formatting or invalid signatures.
- **Error Logging**: Log all protocol errors for debugging and future reference.
- **User Notification**: Provide clear and actionable feedback to users regarding protocol errors to help them correct the issues.

Handling in Practice:

By implementing robust error handling and best practices, you can ensure that transaction errors are minimized and appropriately managed, providing a smoother and more reliable user experience.

8.4 – History Error Handling

Handling errors related to the history of transactions is crucial for maintaining the integrity and reliability of wallet applications. The primary issue you might encounter with the Explorer API is its offline status. Here’s how to effectively manage these errors:

Offline Explorer API

The Explorer API may go offline for restarts or maintenance, especially during database checks and synchronization. When this happens:

1. **Detection**:
    - The service will be unreachable, and connection attempts will time out.
    - Implement a mechanism to detect timeouts and interpret them as potential offline status.
2. **Handling Offline Status**:
    - **Set a Timer**: Block further requests for a predefined period to prevent overwhelming retries.
    - **User Notification**: Inform users that the service is temporarily unavailable and retry later.
    - **Retry Mechanism**: After the timer expires, automatically retry the connection.
3. **Heavy Request Load**:
    - During high load times, responses may be delayed.
    - **User Notification**: Inform users that their request is being processed but may take some time.
    - **Asynchronous Processing**: If possible, handle history requests asynchronously to prevent blocking other operations.

Protocol Errors

Most other errors are likely to be protocol errors, such as malformed requests or querying non-existent wallets. Properly handling these ensures a smoother user experience and accurate error reporting.

1. **Request Validation**:
    - Before sending a request, validate it against the known API schema.
    - Ensure that all required parameters are present and correctly formatted.
2. **Error Logging**:
    - Log detailed information about the error, including the request parameters and the API response.
    - Use these logs for debugging and improving the request handling logic.
3. **User Feedback**:
    - Provide clear and actionable feedback to users regarding the nature of the error.
    - Suggest corrective actions, such as checking the wallet address or parameters.

By implementing these strategies, you can effectively handle history-related errors, ensuring a more robust and user-friendly wallet application.

8.5 – Pending Transactions

Handling pending transactions effectively is crucial for maintaining the reliability and trustworthiness of your wallet application. Here's how you can manage and track pending transactions:

Local-Transaction-ID and Node-Transaction-ID Tracking

To prevent losing track of new pending transactions, especially during node disconnections, maintain a combination of Local-Transaction-ID and Node-Transaction-ID for each signed and validated transaction.

1. **Tracking Mechanism**:
    - Assign a unique Local-Transaction-ID for each transaction initiated by the wallet.
    - Upon receiving a validated transaction from a node, record the associated Node-Transaction-ID.
2. **Reconnection Handling**:
    - If a node disconnects while a transaction is pending, use the recorded Node-Transaction-ID to query other nodes or the explorer API for the transaction status.
    - This ensures that you can continue tracking the transaction even after switching nodes.
3. **Explorer API Check**:
    - Utilize the explorer API to verify the status of transactions using their Node-Transaction-ID.
    - Example API query: `explorer?command=get&transid=[transid]` returns either an undefined or a seal-object (processed transaction).

Node Disconnection Scenario

When a node disconnects after validating but before processing a transaction, your wallet might not receive the confirmation event. Here's how to handle this:

1. **Track Transaction IDs**:
    - Maintain a list of pending transactions with their Node-Transaction-IDs.
    - Periodically check these IDs against the explorer API to see if they have been processed.
2. **Update Wallet Status**:
    - If the explorer API confirms a transaction as processed, update your wallet accordingly.
    - If a transaction remains pending for an extended period, consider retrying or canceling the transaction.

Future Enhancement: NodeLeaf-API `transstatus`

The '`transstatus'` command, which is on the development roadmap, will provide additional functionality for tracking transactions. It will offer:

1. **Transaction Status Check**:
    - Query the status of a specific transaction by its ID.
    - Parameters: `command:'transstatus', tid:'[transid]'` returning status as undefined, pending, or processed.

2. **Pending Transactions List**:
    - Retrieve a list of all pending transactions for a specific wallet address.
    - Parameters: `command:'transstatus", wallet:'[wallet]'` returning a list of transaction IDs.

Once implemented, this command will greatly enhance the ability to track and manage pending transactions.

By following these best practices and preparing for future enhancements, you can ensure robust handling of pending transactions in your wallet application, providing a reliable and secure user experience.

8.6 – History Update Event Triggers

Handling history update event triggers efficiently ensures that your wallet application maintains an up-to-date view of processed transactions and wallet balances without generating unnecessary overhead. Here's a structured approach to manage these updates:

Event Trigger Flow

The event trigger flow for processed transactions, wallet balance changes, and history updates involves the following steps:

1. **Transaction Processing**:
    - When a transaction is successfully processed, it triggers a series of events including updates to the wallet balance and transaction history.
2. **Balance Change Detection**:
    - After a transaction, detect the change in wallet balance using node API calls.
    - If a balance change is detected, it indicates a new transaction has been processed.
3. **History Update Timer**:
    - To avoid frequent and redundant API calls, set a timer for one minute before querying the explorer API for a history update.
    - This timer is based on the explorer's update frequency, which syncs the ledger from the nodes every minute.

Implementation Details

1. **Balance Change Event**:
    - Use the nodeLeaf-API to check for balance changes. If a change is detected, it implies a new transaction has been processed.
2. **History Update Timer**:
    - Upon detecting a balance change, set a timer for one minute.
    - After the timer expires, query the explorer API to update the transaction history.
3. **Handling CoinBase and Fee Payouts**:

- Similar to regular transactions, coinbase and fee payouts also trigger balance changes.
- The same timer and update mechanism apply to these events.

Summary

By following these best practices, you ensure that your wallet application:

- Detects balance changes promptly.
- Updates transaction history efficiently with minimal overhead.
- Maintains an accurate and up-to-date view of wallet activity.

This approach balances the need for timely updates with the goal of reducing unnecessary API calls, ensuring a smooth and efficient user experience.

9. Security Considerations

Overall, we handle only a wallet file containing the private keys that enable transaction signatures of claimed outblocks from genesis, coinbase, fee, or regular transaction records within the ledger of the nodes. These private keys are crucial as they authorize transactions and secure the user's assets. The communication protocols do not include any private data elements in a direct sense. Instead, they focus on facilitating secure and efficient transactions.

Transactions are uniquely signed by the user's wallet, ensuring that only the rightful owner can authorize movements of funds. To maximize security, we implement robust measures to protect the private area of the wallet, even in private locations. This includes encrypting wallet files with a strong password, which is required to decrypt and access the wallet's functionalities.

By securing the wallet with a decrypting password, we provide users with the assurance that their assets are protected from unauthorized access, while still enabling them to utilize all the features and functions associated with their wallet. This approach ensures that even if the wallet file is compromised, the private keys remain secure, maintaining the integrity and security of the user's transactions and assets within the blockchain ecosystem.

9.1 – Secure Storage of Private Keys

Given the paramount importance of private key security, we implement rigorous measures to ensure these keys are heavily protected, particularly during communication. To safeguard private keys, we recommend encrypting every export with a user-defined password. This password is essential for encoding and decoding the private data, ensuring that only the user can access their sensitive information.

As a best practice, we ensure that private keys are exported exclusively to the user's devices. We do not share these keys with other wallet providers, such as exchanges. Instead, exchanges must create their own wallets to which users can transfer their cryptocurrency from their private wallets.

Wallet providers bear the responsibility of establishing and maintaining their security standards to protect users' assets. Our assurance extends only to the core FactorialCoin system we have

designed, which is purpose-built to provide a secure and robust foundation for managing cryptocurrency transactions.

9.2 – Use of HTTPS and WebSocket

For centralized communications, we prefer using the well-established HTTPS (HTTP Secure) standard to ensure data integrity and privacy. This standard is widely recognized for its effectiveness in protecting data transmitted over the internet.

For communication between nodes, we utilize the WebSocket protocol without an additional SSL layer. This decision is based on the nature of the data being transmitted, which does not include private information. By forgoing SSL, any IP address can be utilized as a node IP address, facilitating broader participation in the network. Transactions are initiated through signature authentications, ensuring that only valid actions are processed. These transactions ultimately end up in a public ledger, and wallet balance replies are also publicly accessible.

Additionally, miners deliver their encrypted solutions to the network in a way that ensures only the coinbase server can decode them. The miner encrypts their wallet address with the solution, a method that only the miner and the coinbase server understand. This process enhances security by preventing nodes from altering the wallet address associated with a solution before it reaches the coinbase server.

9.3 – Regular Audits

Regular audits will become a standard practice in the future as more stakeholders show interest in ensuring the system's integrity. We aim to establish a structure that is self-scrutinizing, promoting transparency and accountability. However, to ensure the utmost reliability of software versions, we welcome external parties to examine every step of the process. These audits can encompass both the core system and the ancillary systems connected to it, ensuring that each decentralized component operates securely and robustly.

To provide the best user experience, we prioritize the core functionalities of the system, maintaining a stable foundation for all user software interacting with the network. This focus on the core ensures that the entire ecosystem remains secure, efficient, and reliable, supporting the smooth operation of user interfaces and interactions with the network.

10. Conclusion

10.1 – Summary of Key Points

Recap of Wallet Management

Wallet management is central to the security and functionality of the FactorialCoin ecosystem. The wallet system relies on a file containing a collection of wallets with private keys that enable transaction signatures for transactions within the ledger and validated by the nodes. The encryption

of these keys is paramount, ensuring that they are protected with a decrypting password to secure the user's access to functionalities. By emphasizing strong encryption practices and secure storage, we ensure that users' private keys remain confidential and safe from unauthorized access. See the `fccencode` function for details of how to protect and encode/decode a privatekey in the `savewallets` function which stores the wallets data on disk in a wallet file called `wallet.fcc` in FCC's case.

Overview of API Integration

FactorialCoin employs WebSocket for real-time, bidirectional communication, JSON for standardized data exchange, and HTTPS for secure HTTP API access. This setup ensures efficient, reliable, and secure interactions between wallets, miners, and network nodes. The API facilitates seamless integration, enabling developers to build robust applications while maintaining high security standards. By providing comprehensive API documentation and support, we enable developers to effectively utilize FactorialCoin's features, fostering innovation and expanding the ecosystem.

Importance of Security Considerations

Security is a cornerstone of FactorialCoin. We implement multiple layers of protection:

- **Encryption of Wallets and Miner Solutions:** Solutions are encrypted with wallet information using Ed25519 to prevent tampering with transactions. Miners are ensured that only the coinbase server can decode the wallet address associated with the solution, maintaining security during the mining process.
- **Secure Communication:** WebSocket and JSON over HTTPS ensure encrypted and secure data transmission, protecting sensitive information from unauthorized access. While WebSocket is used for bidirectional communication and JSON for standard data exchange, HTTPS is utilized for HTTP API access, ensuring privacy and integrity.
- **Audits and Protocols:** Regular audits and adherence to security protocols fortify the system against vulnerabilities, ensuring a robust and secure blockchain ecosystem. We plan to establish a self-scrutinizing structure and engage third-party audits to validate the security and functionality of the system.

Examples and Best Practices

- **Connection Error Handling:** Implementing backup nodelists and rotating nodes to maintain connectivity during service disruptions.
- **Balance Error Handling:** Addressing balance errors arising from unlisted or invalid wallet addresses.
- **Transaction Error Handling:** Preventing failed transactions by managing wallet actions on the same node and handling potential double-spending scenarios.
- **History Error Handling:** Managing offline Explorer API scenarios with local buffers and timers to update wallet history once the service is back online.

- **Pending Transactions:** Tracking transactions using Local-Transaction-ID and Node-Transaction-ID combinations to ensure proper handling and confirmation of pending transactions.
- **History Update Event Triggers:** Implementing timers for balance and history updates to minimize overhead and ensure accurate wallet information.

10.2 – Future Enhancements

Potential Developments in the FactorialCoin Ecosystem

Looking ahead, we anticipate several advancements within the FactorialCoin ecosystem:

- **Decentralized Explorer API:** Plans to split the Explorer API from the central server and operate it over WebSocket for decentralized access, reducing load and enhancing performance. This decentralized option will also support history update push messages, eliminating the need for periodic polling.
- **Enhanced Node Commands:** Implementation of commands like `transstatus` to track pending transactions, providing more transparency and control over transaction states. This will allow users to check the status of their transactions and ensure they are processed correctly.
- **Scalability Improvements:** Enhancing the network to handle a growing number of transactions and users while maintaining performance and security.
- **Integration with Other Community Coin Developments:** Exploring interoperability with other community coin developments to create a wider decentralized network of ledgers. This will allow us to join and collaborate on the FactorialCoin Exchange (FcEx), facilitating cross-chain transactions and broadening FactorialCoin's applicability.
- **Exchange Pair Connections:** Establishing exchange pair connections with other prominent cryptocurrencies to enhance liquidity and trading options on the FactorialCoin Exchange (FcEx). This initiative aims to foster seamless trading experiences and expand the utility of FactorialCoin across diverse cryptocurrency markets. The new wallet version will include API trading capabilities, enabling users to execute trades directly through the FactorialCoin wallet interface, thereby simplifying access to cryptocurrency markets and enhancing user convenience.

Encouraging Community Growth and Contributions

Community engagement is vital for the continuous improvement and decentralization of FactorialCoin. We encourage contributions from developers, researchers, and users to:

- **Participate in Audits:** Help scrutinize and validate the system, ensuring its integrity and reliability. Community-led audits will enhance transparency and trust in the FactorialCoin ecosystem.
- **Develop Applications:** Build new tools and applications that leverage FactorialCoin’s secure and efficient protocols. By providing development resources and support, we aim to foster innovation and expand the ecosystem.

- **Foster Innovation:** Explore and implement novel features and enhancements that contribute to the ecosystem's growth and robustness. We welcome ideas and projects that push the boundaries of what FactorialCoin can achieve.
- **Engage in Governance:** Participate in the decision-making process to shape the future of FactorialCoin, ensuring that it meets the needs and expectations of its diverse user base.

By fostering a collaborative environment and continually advancing our technology, we aim to create a secure, efficient, and user-friendly blockchain platform that meets the evolving needs of the digital economy. Our commitment to privacy, security, transparency, and community involvement will drive the ongoing development and success of the FactorialCoin ecosystem.

11. Appendices

11.1 – Leaf API Reference

The Leaf API Reference provides comprehensive documentation on the API endpoints and communication protocols essential for interacting with FactorialCoin's blockchain network. This section outlines the methods and interactions necessary for developers and integrators to effectively interface with FactorialCoin's nodes.

Introduction to FactorialCoin Leaf API

FactorialCoin's Leaf API operates primarily over WebSocket for real-time interactions, complemented by HTTP for initial setup and node list retrieval. It utilizes JSON for data exchange and employs Ed25519 encryption to secure sensitive transaction data and communications.

API Overview

- **WebSocket Support**: All communications within the Leaf API occur over WebSocket, ensuring efficient and real-time data transmission between clients and nodes.
- **JSON Communications**: Data exchanges throughout the API are formatted in JSON, providing a standardized and lightweight protocol for transmitting information across the network.
- **Ed25519 Encryption**: Encryption plays a crucial role in securing sensitive data, such as transaction signatures. FactorialCoin employs the Ed25519 cryptographic algorithm to ensure robust authentication and data integrity.

Node List

FactorialCoin maintains a dynamic list of active nodes available for connection, facilitating network resilience and load distribution.

- **FCC Node List**: Accessible at https://factorialcoin.nl:5151/?nodelist, this resource provides details of active FactorialCoin nodes available for connection.

Wallet-Leaf Connection Protocol

The interaction between client wallets (Leaf) and FactorialCoin's nodes involves several key commands and responses, facilitating secure and efficient transaction processing:

1. **hello**
    - **Incoming**: Upon establishing a connection, the FactorialCoin node sends a `hello` message to the Leaf API client, initiating the identification process.
    - **Outgoing**: The Leaf API client responds with an `identify` command, specifying its type (leaf) and the ledger version from the node.
2. **balance**
    - **Outgoing**: Requests the current balance of a specified FCC wallet address.
    - **Incoming on Success**: Returns the current balance associated with the provided wallet address.
    - **Incoming on Error**: Provides an error message if the balance retrieval operation encounters issues.
3. **newtransaction**
    - **Outgoing**: Initiates a new transaction request with details including transaction ID, recipient wallet address, amount, and transaction fee.
    - **Incoming on Success**: Provides the transaction ID and necessary data for client-side signing of the transaction.
    - **Incoming on Error**: Communicates an error message if the transaction submission process encounters issues.
4. **signtransaction**
    - **Outgoing**: Signs the transaction data using the client's private key, confirming the validity and authenticity of the transaction request.
    - **Incoming on Success**: Confirms the successful signing of the transaction with the node-generated transaction ID.
    - **Incoming on Error**: Indicates an error message if the transaction signing process fails.
5. **processed**
    - **Incoming on Success**: Notifies the client of a successfully processed transaction, providing details such as the transaction hash, associated wallet address, and status of the transaction.
    - **Incoming on Error**: Reports an error message if the transaction processing encounters issues.

Miner-Leaf Connection Protocol

For miners engaged in mining activities within FactorialCoin's blockchain network, the protocol supports mining operations:

1. **hello**

- **Incoming**: Upon establishing a connection, the FactorialCoin node sends a `hello` message to the Leaf API client, initiating the identification process.
- **Outgoing**: The Leaf API client responds with an `identify` command, specifying its type (miner) and version.

2. **mine**
    - **Incoming**: Upon identification as a miner, the node sends a `mine` command to initiate the mining process.
    - **Outgoing**: The miner client responds with mined solutions, including relevant details such as solution hash and associated wallet address.
3. **solution**
    - **Outgoing**: Sends mined solutions from the miner client to the node, including details such as solution hash and associated wallet address.
    - **Incoming on Error**: Indicates an error if the solution submission encounters issues.
    - **Incoming on Success**: Confirms successful receipt and processing of the mined solution by the node.

Conclusion

The FactorialCoin Leaf API Reference offers a structured overview of the communication protocols and API endpoints necessary for integrating applications with FactorialCoin's blockchain network. Developers and system integrators can utilize this documentation to build robust, secure, and scalable solutions that seamlessly interact with the FactorialCoin ecosystem, facilitating efficient blockchain transactions and mining operations.

11.2 – Original Perl Code Snippets

Introduction to FCC::global.pm Module

The **FCC::global.pm** module serves as a foundational component within the broader system architecture of the FCC (Financial Cryptocurrency) project. This module encapsulates essential constants, variables, and utility functions that are fundamental to various operations and functionalities across the FCC ecosystem.

Constants and Configuration

1. **Cryptographic Constants**:
    - **$COIN**, **$FCCVERSION**, **$FCCBUILD**: Define the coin type, FCC version, and build information respectively.
    - **$FCCMAGIC**: Identifies the magic number for protocol validation.
2. **Server Configuration**:
    - **$FCCSERVERIP**, **$FCCSERVERHOST**, **$FCCSERVERPORT**: Specify server IP address, hostname, and port number.
    - **$FCCSERVERKEY**: Stores the server's public key for cryptographic operations.

3. **Transaction and Fee Management**:
    - **$MINIMUMFEE**: Sets the minimum transaction fee.
    - **$MINERPAYOUT**: Determines miner payout configuration.
    - **$MINEBONUS**: Applies a bonus amount for mining operations.
4. **Transaction Types**:
    - **$TRANSTYPES**, **$RTRANSTYPES**: Store transaction types and their reverse mappings respectively.
5. **Hexadecimal Operations**:
    - **%HP**: Provides mappings for hexadecimal characters to decimal values.

Utility Functions

- **setcoin**: Configures the coin type and updates associated constants.
- **tzoffset**: Computes the timezone offset in seconds.
- **fcctime**, **setfcctime**, **fcctimestring**: Manage FCC-specific time operations, including formatting and setting time.
- **securehash**: Computes secure hashes using SHA-512 and SHA-256 algorithms for data integrity and security.
- **octhex**, **hexoct**, **hexchar**, **dechex**, **hexdec**: Convert between octets, hexadecimal strings, and decimal values.

Base64 Encoding and Decoding

- **validh64**: Validates a base64 encoded string.
- **encode_base64**: Encodes data to base64 format.
- **decode_base64**: Decodes base64 encoded data.

Response and Fee Calculation

- **rsp**: Generates standard HTML response templates for server interactions.
- **calcfee**: Calculates transaction fees based on transaction size.
- **doggyfee**: Computes specific transaction fees with additional bonuses.

FCC String and Encoding

- **fccstring**: Creates an FCC-specific string using octal and hexadecimal conversions.
- **fccencode**, **zb64**: Encodes strings using ZB64 encoding scheme.
- **b64z**: Decodes data encoded with B64Z format.

Data Compression

- **zip**: Compresses data using zlib compression algorithm.
- **unzip**: Decompresses zlib-compressed data.

The FCC::global.pm package

```
perl
```

```
#!/usr/bin/perl

package FCC::global;

use strict;
use warnings;
use Exporter;
use vars qw($VERSION @ISA @EXPORT @EXPORT_OK);

# Module version and Exporter setup
$VERSION     = '2.3.2';
@ISA         = qw(Exporter);
@EXPORT      = qw($COIN $HP setcoin $FCCVERSION $FCCBUILD $FCCEXT $FCCTIME $FCCMAGIC $FCCSERVERKEY
$TRANSTYPES $RTRANSTYPES
                  $MINIMUMFEE $MINERPAYOUT $MINEBONUS $FCCSERVERIP $FCCSERVERHOST $FCCSERVERPORT
ledgerversion
                  prtm securehash octhex hexoct hexchar dechex hexdec validh64 encode_base64
decode_base64 rsp
                  fcctime setfcctime fcctimestring extdec doggy calcfee doggyfee fccstring fccencode
zb64 b64z zip unzip);
@EXPORT_OK   = qw();

# Global constants
our $COIN = "FCC";  # Default coin
our $FCCVERSION = "0101"; # Ledger version
our $FCCBUILD = $VERSION;   # Software version
our $FCCTIME = tzoffset();  # Timezone offset
our $FCCMAGIC = 'FF2F89B12F9A29CAB2E2567A7E1B8A27C8FA9BF7A1ABE76FABA7919FC6B6FF0F';  # Magic number
our $FCCSERVERIP = '149.210.194.88';  # Server IP address
our $FCCSERVERHOST = 'factorialcoin.nl';  # Server hostname
our $FCCSERVERPORT = 5151;  # Server port
our $FCCSERVERKEY = "FCC55202FF7F3AAC9A85E22E6990C5ABA8EFBB73052F6EA1867AF7B96AE23FCC";  # Server
key
our $FCCEXT = '.fcc';  # File extension
our $MINIMUMFEE = 50;  # Minimum transaction fee
our $MINERPAYOUT = 1000000000;  # Minimum miner payout
our $MINEBONUS = 50000000;  # Miner bonus amount

# Transaction types
our $TRANSTYPES = {
    genesis => '0',
    in => '1',
    out => '2',
    coinbase => '3',
    fee => '4'
};

# Reverse transaction types
our $RTRANSTYPES = {};
foreach my $k (keys %$TRANSTYPES) {
    $RTRANSTYPES->{$TRANSTYPES->{$k}} = $k
}

# Hexadecimal to decimal mapping for characters A-F
our $HP = {};
for (my $i=0; $i<10; $i++) { $HP->{$i} = $i }
$HP->{'A'} = 10; $HP->{'B'} = 11; $HP->{'C'} = 12; $HP->{'D'} = 13; $HP->{'E'} = 14; $HP->{'F'} =
15;

# Function to set the coin type
sub setcoin {
    $COIN = uc($_[0]);
    if ($COIN eq 'PTTP') {
        $FCCMAGIC = "8BF879BEC8FA9EC6CA3E7A96B26F7AA76F6AA4E78BADCFA1665A8A9CD67ADD0F";
        $FCCSERVERPORT = 9612;
        $FCCSERVERKEY = "1111145AFA4FBB1CF8D406A234C4CC361D797D9F8F561913D479DBC28C7A4F3E";
        $FCCEXT = '.pttp';
        $FCCBUILD = '1.4.2';
        $MINIMUMFEE = 110;
    } elsif ($COIN eq 'FCC') {
        $FCCMAGIC = "FF2F89B12F9A29CAB2E2567A7E1B8A27C8FA9BF7A1ABE76FABA7919FC6B6FF0F";
        $FCCSERVERPORT = 5151;
        $FCCSERVERKEY = "FCC55202FF7F3AAC9A85E22E6990C5ABA8EFBB73052F6EA1867AF7B96AE23FCC";
        $FCCEXT = '.fcc';
        $FCCBUILD = $VERSION;
        $MINIMUMFEE = 50;
```

```
    } elsif ($COIN ne 'FCC') {
        die "Unknown coin '$_[0]'"
    }
}

# Function to calculate the timezone offset
sub tzoffset {
    my $t = time();
    my $utc = mktime(gmtime($t));
    my $local = mktime(localtime($t));
    return ($utc - $local);
}

# Function to set the FCC time
sub fcctime {
    if (!$_[0]) { $FCCTIME = 0; return }
    my $t = time();
    my $local = mktime(localtime($t));
    $FCCTIME = $_[0] - $local;
}

# Function to get the ledger version
sub ledgerversion {
    my $major = int substr($FCCVERSION, 0, 2);
    my $minor = int substr($FCCVERSION, 2, 2);
    return join('.', $major, $minor);
}

# Function to set the FCC time directly
sub setfcctime {
    $FCCTIME = $_[0]
}

# Function to format FCC time into a string
sub fcctimestring {
    my ($time) = @_;
    if (!$time) { $time = time() + $FCCTIME }
    my @t = localtime($time);
    my $tm = ('Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat')[$t[6]] . ", ";
    my $yr = $t[5] + 1900;
    my $mon = ('Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec')
[$t[4]];
    $tm .= "$t[3] $mon $yr ";
    $tm .= join(':', sprintf("%02d", $t[2]), sprintf("%02d", $t[1]), sprintf("%02d", $t[0]));
    $tm .= " GMT";
    return $tm;
}

# Function to generate a secure hash
sub securehash {
    my ($code) = @_;
    if (!$code) { error "FCC.Global.SecureHash: No Code given to hash!" }
    return uc(sha256_hex(sha512_hex($code)));
}

# Function to convert octets to hexadecimal
sub octhex {
    my ($key) = @_;
    if (!defined $key) { return "" }
    my $hex;
    for (my $i=0; $i<length($key); $i++) {
        my $c = ord(substr($key, $i, 1));
        $hex .= sprintf('%02X', $c);
    }
    return $hex;
}

# Function to convert hexadecimal to octets
sub hexoct {
    my ($hex) = @_;
    if (!defined $hex) { return "" }
    my $key = "";
    for (my $i=0; $i<length($hex); $i+=2) {
        my $h = substr($hex, $i, 2);
        $key .= chr(hex($h));
    }
    return $key;
```

```
}

# Function to convert a number to its hexadecimal character representation
sub hexchar {
    my ($num) = @_;
    return (0,1,2,3,4,5,6,7,8,9,'A','B','C','D','E','F')[$num];
}

# Function to convert decimal to hexadecimal
sub dechex {
    my ($dec, $len) = @_;
    if (!defined $dec) { error ("FCC::global::dechex: No decimal given") }
    if (!$len) { error "FCC::global::dechex - No length given" }
    my $out = "";
    if ($len == 1) { return hexchar($dec & 15) }
    while ($len > 0) {
        my $byte = $dec & 255;
        my $hex = hexchar($byte >> 4);
        $hex .= hexchar($byte & 15);
        $out = "$hex$out";
        $dec >>= 8;
        $len -= 2;
    }
    return $out;
}

# Function to convert hexadecimal to decimal
sub hexdec {
    my ($hex) = @_;
    if ($hex =~ /[^0-9A-F]/) {
        error "FCC::global::hexdec - Illegal hex given '$hex'";
    }
    my $dec = 0;
    for (my $i = 0; $i < length($hex); $i++) {
        $dec = $dec * 16 + $HP->{substr($hex, $i, 1)};
    }
    return $dec;
}

# Function to validate a base64 encoded string
sub validh64 {
    my ($h64) = @_;
    if (!$h64) { return "" }
    if ($h64 =~ /[^0-9A-Za-z+\/]/) {
        error "FCC::global::validh64 - illegal h64"
    }
    return 1;
}

# Function to encode data to base64
sub encode_base64 {
    my ($data) = @_;
    if (!$data) { error "FCC::global::encode_base64 - No data to encode!" }
    return MIME::Base64::encode_base64($data);
}

# Function to decode base64 encoded data
sub decode_base64 {
    my ($b64) = @_;
    if (!$b64) { error "FCC::global::decode_base64 - No B64 to decode!" }
    return MIME::Base64::decode_base64($b64);
}

# Function to respond with a standard template
sub rsp {
    my ($code, $html) = @_;
    if ($code ne '401') { $code = '200' }
    return "<HTML>\n<HEAD><TITLE>FCC: $code</TITLE></HEAD>\n<BODY>$html\n</BODY>\n</HTML>\n";
}

# Function to calculate the fee for a transaction
sub calcfee {
    my ($size) = @_;
    if ($size < 1000) { $size = 1000 }
    my $fee = int($size / 1000);
    if ($size % 1000) { $fee++ }
    if ($fee < $MINIMUMFEE) { $fee = $MINIMUMFEE }
```

```
    return $fee;
}

# Function to calculate the doggy fee
sub doggyfee {
    my ($size) = @_;
    if ($size < 1000) { $size = 1000 }
    my $fee = int($size / 1000);
    if ($size % 1000) { $fee++ }
    if ($fee < $MINIMUMFEE) { $fee = $MINIMUMFEE }
    $fee += $MINEBONUS;
    return $fee;
}

# Function to create a FCC string
sub fccstring {
    my ($str) = @_;
    return "FCC" . octhex($str);
}

# Function to encode a string using ZB64
sub fccencode {
    my ($str) = @_;
    return zb64($str);
}

# Function to encode using ZB64
sub zb64 {
    my ($data) = @_;
    my $b64 = encode_base64($data);
    $b64 =~ s/\+/\$/g; $b64 =~ s/\//\./g; $b64 =~ s/=//g;
    return $b64;
}

# Function to decode using B64Z
sub b64z {
    my ($b64) = @_;
    $b64 =~ s/\$/\+/g; $b64 =~ s/\./\//g;
    my $data = decode_base64($b64);
    return $data;
}

# Function to zip data
sub zip {
    my ($data) = @_;
    return Compress::Zlib::memGzip($data);
}

# Function to unzip data
sub unzip {
    my ($data) = @_;
    return Compress::Zlib::memGunzip($data);
}

1;
```

Explanation of FCC::global.pm

- **Constants and Variables:**
  - `$COIN`, `$FCCVERSION`, `$FCCBUILD`, `$FCCTIME`, `$FCCMAGIC`, `$FCCSERVERIP`, `$FCCSERVERHOST`, `$FCCSERVERPORT`, `$FCCSERVERKEY`, `$FCCEXT`, `$MINIMUMFEE`, `$MINERPAYOUT`, `$MINEBONUS`, `$TRANSTYPES`, `$RTRANSTYPES`, `$HP`: Global constants and variables used throughout the module for configuration, transaction types, server details, and cryptographic operations.
- **Functions:**
  - `setcoin`: Sets the coin type and updates associated constants based on the coin type.

- `tzoffset`: Calculates the timezone offset in seconds.
- `fcctime`, `setfcctime`, `fcctimestring`: Functions related to managing and formatting FCC time.
- `securehash`: Computes a secure hash using SHA-512 and SHA-256.
- `octhex`, `hexoct`, `hexchar`, `dechex`, `hexdec`: Conversion functions between octets, hexadecimal strings, and decimal values.
- `validh64`, `encode_base64`, `decode_base64`: Functions for base64 validation, encoding, and decoding.
- `rsp`: Generates an HTML response template.
- `calcfee`, `doggyfee`: Calculate transaction fees based on transaction size.
- `fccstring`, `fccencode`, `zb64`, `b64z`: Functions for encoding and decoding using ZB64.
- `zip`, `unzip`: Functions for compressing and decompressing data using zlib.

Summary

The **FCC::global.pm** module is pivotal within the FCC system, providing centralized management of constants, configurations, and utility functions essential for cryptography, data encoding, transaction processing, and server interactions. It establishes a reliable foundation for other modules, ensuring consistent and secure operations throughout the FCC application ecosystem.

Introduction to FCC::wallet.pm Package

The **FCC::wallet.pm** package, part of the FCC (Financial Cryptocurrency) project, focuses on managing cryptocurrency wallets securely. Developed by Domero in 2019, this module enhances the FCC system by providing robust wallet functionalities built on Perl.

Constants and Configuration

- **$WALLETDIR**: Default directory for storing wallet files.
- **$WALLETEXISTS**: Checks if the wallet file exists.
- **@WXOR**: Array used for checksum calculation.
- **$COIN**: Specifies the cryptocurrency type (e.g., 'PTTP').

Utility Functions

- **findwallet**: Locates the wallet file in different possible directories.
- **publichash**: Retrieves the public hash key from a wallet.
- **validatehash**: Validates a wallet identifier against a public key.
- **createwalletaddress**: Generates a new wallet address based on a public key.
- **newwallet**: Creates a new wallet with a name, public and private keys.
- **validwallet**: Validates the format and checksum of a wallet address.
- **walletexists**: Checks if a wallet file exists in the specified directory.
- **walletisencoded**: Checks if the wallet data is encoded for security.
- **loadwallets**: Loads wallets from a file, optionally decrypting with a password.
- **savewallet**: Saves a wallet to the wallet file.

- **savewallets**: Saves multiple wallets to the wallet file.
- **loadwallet**: Loads a specific wallet from the wallet file.
- **validwalletpassword**: Validates the password for accessing the encrypted wallet.

Cryptographic and Data Handling

- **securehash**: Computes a secure hash of input data.
- **fccencode**, **fccdecode**: Encodes and decodes data for wallet security.
- **octhex**, **hexoct**: Converts hexadecimal data to octal and vice versa.
- **dec_json**, **toJSON**: Functions for decoding and encoding JSON data.

The FCC::wallet.pm Package

```
perl

#!/usr/bin/perl

package FCC::wallet;

########################################
#                                      #
#     FCC Wallet                       #
#                                      #
#    (C) 2019 Domero                   #
#                                      #
########################################

use strict;
use warnings;
use Exporter;
use vars qw($VERSION @ISA @EXPORT @EXPORT_OK);

$VERSION     = '2.1.5';
@ISA         = qw(Exporter);
@EXPORT      = qw($WALLETEXISTS $WALLETDIR
                  publichash validatehash createwalletaddress walletexists walletisencoded
validwalletpassword
                  newwallet validwallet loadwallet loadwallets savewallet savewallets);
@EXPORT_OK   = qw();

use gfio 1.11;
use gerr 1.02;
use Crypt::Ed25519;
use glib;
use FCC::global 2.2.1;

our $WALLETDIR=".";  # Default wallet directory
our $WALLETEXISTS=&findwallet();  # Check if wallet exists

my @WXOR = ();  # XOR table for checksum calculation
createtable();  # Create XOR table

1;

# Wallet structure
#
# offset length  content
#      0      2  '51' - FCC identifier (11 for PTTP)
#      2     64  Public hashkey
#     66      2  Checksum, xor ascii values 0-65 must be 0
#
# Wallet will be converted to uppercase always!

# Create XOR table for checksum calculation
sub createtable {
    my @l=();
    for (my $c=0; $c<10; $c++) { push @l,ord($c) }
    for (my $c='A'; $c le 'F'; $c++) { push @l,ord($c) }
    foreach my $m (@l) {
        foreach my $n (@l) {
```

```
            push @WXOR,{ add => chr($m).chr($n), value => $m ^ $n };
        }
    }
}

# Find wallet file
sub findwallet {
    if (!-f "$WALLETDIR/wallet$FCCEXT") {
        if (-f "$WALLETDIR/wallet/wallet$FCCEXT") {  $WALLETDIR="$WALLETDIR/wallet" }
        elsif (-d "$WALLETDIR/wallet") {  $WALLETDIR="$WALLETDIR/wallet" }
        elsif (-f "../wallet$FCCEXT") {  $WALLETDIR=".." }
        elsif (-f "../wallet/wallet$FCCEXT") { $WALLETDIR="../wallet" }
        elsif (-d "../wallet") {  $WALLETDIR="../wallet" }
    }
    return (-e "$WALLETDIR/wallet$FCCEXT")
}

# Get public hash from wallet
sub publichash {
    my ($wallet) = @_;
    if (ref($wallet) eq "FCC::wallet") { $wallet=$wallet->{wallet} }
    if (validwallet($wallet)) {
        return substr($wallet,2,64)
    }
    return ""
}

# Validate wallet address with public key
sub validatehash {
    my ($wid,$pubkey) = @_;
    if (createwalletaddress($pubkey) eq $wid) {
        return 1
    }
    return 0
}

# Create wallet address from public key
sub createwalletaddress {
    my ($pubkey) = @_;
    my $pubhash=securehash($pubkey);  # Compute secure hash
    my $xor=ord('5') ^ ord('1');  # Initial XOR value
    if ($COIN eq 'PTTP') {
        $xor=ord('1') ^ ord('1');  # Adjust XOR value for PTTP
    }
    for (my $c=0;$c<64;$c++) {
        $xor ^= ord(substr($pubhash,$c,1));  # Update XOR value
    }
    my $checksum="";
    foreach my $try (@WXOR) {
        if (($try->{value} ^ $xor) == 0) {
            $checksum=$try->{add}; last
        }
    }
    if ($COIN eq 'PTTP') {
        return '11'.$pubhash.$checksum;  # Generate wallet address for PTTP
    } else {
        return '51'.$pubhash.$checksum;  # Generate wallet address for FCC
    }
}

# Generate new wallet
sub newwallet {
    my ($name) = @_;
    if (!$name) { $name = "[ No name ]" }
    my ($pubkey, $privkey) = Crypt::Ed25519::generate_keypair;  # Generate keypair
    my $pubhex = octhex($pubkey);  # Convert public key to hex
    my $wallet = {
        pubkey => $pubhex,
        privkey => octhex($privkey),
        wallet => createwalletaddress($pubhex),
        name => $name
    };
    bless($wallet); return $wallet  # Bless and return wallet object
}

# Validate wallet format
sub validwallet {
```

```
    my ($wallet) = @_;
    if (!$wallet) { return 0 }
    $wallet=uc($wallet);
    if (length($wallet) != 68) { return 0 }
    my $xor=ord('5') ^ ord('1');  # Initial XOR value
    if ($COIN eq 'PTTP') {
        $xor=ord('1') ^ ord('1');
        if (substr($wallet,0,2) ne '11') { return 0 }
    } else {
        if (substr($wallet,0,2) ne '51') { return 0 }
    }
    for (my $c=2;$c<68;$c++) {
        my $h=substr($wallet,$c,1);
        if ((($h ge '0') && ($h le '9')) || (($h ge 'A') && ($h le 'F'))) {
            $xor ^= ord($h)
        } else {
            return 0
        }
    }
    if ($xor != 0) { return 0 }
    return 1
}

# Check if wallet file exists
sub walletexists {
    return (-e "$WALLETDIR/wallet$FCCEXT")
}

# Check if wallet file is encoded
sub walletisencoded {
    if (-e "$WALLETDIR/wallet$FCCEXT") {
        my $winfo=dec_json(gfio::content("$WALLETDIR/wallet$FCCEXT"));
        if (ref($winfo) eq 'HASH') {
            if ($winfo->{encoded}) { return 1 }
        }
    }
    return 0
}

# Load wallets from file
sub loadwallets {
    my ($password) = @_;
    my $wlist=[];
    print "Looking for wallet at $WALLETDIR/wallet$FCCEXT\n";
    if (-e "$WALLETDIR/wallet$FCCEXT") {
        my $winfo=dec_json(gfio::content("$WALLETDIR/wallet$FCCEXT"));
        if (ref($winfo) eq 'HASH') {
            # Handle wallet version 2+
            if ($winfo->{encoded}) {
                my ($seed,$hash)=(substr($winfo->{encoded},0,8),substr($winfo->{encoded},8));
                if (securehash($seed.$COIN.$password) ne $hash) { return [ { error => 'invalid
password' } ] }
            }
            $wlist=$winfo->{wlist};
            foreach my $wallet (@$wlist) {
                bless($wallet);  # Bless each wallet object
                if (!$wallet->{name}) { $wallet->{name}="[ No name ]" }
                if ($winfo->{encoded}) {
                    if ($wallet->{pubkey}) { $wallet->{pubkey}=fccencode(hexoct($wallet->{pubkey}),
$password) }
                    if ($wallet->{privkey}) { $wallet->{privkey}=fccencode(hexoct($wallet-
>{privkey}),$password) }
                    for my $type ('private','public','contact') {
                        if (ref($wallet->{$type}{Debit})) {
                            for my $c (keys %{$wallet->{$type}{Debit}}) {
                                for my $d (@{$wallet->{$type}{Debit}{$c}}) {
                                    if ($d->{pubkey}) { $d->{pubkey}=fccencode(hexoct($d->{pubkey}),
$password) }
                                    if ($d->{privkey}) { $d->{privkey}=fccencode(hexoct($d-
>{privkey}),$password) }
                                }
                            }
                        }
                    }
                }
            }
        } else {
```

```
            # Handle wallet version 1
            foreach my $wallet (@$wlist) {
                if (!$wallet->{name}) { $wallet->{name}="[ No name ]" }
            }
            $wlist=$winfo
        }
    }
    return $wlist
}

# Save wallet to file
sub savewallet {
    my ($wallet,$password) = @_;
    if (ref($wallet) ne "FCC::wallet") { error "FCC::wallet::savewallet - Wallet given is not a FCC
blessed wallet" }
    my $wlist=loadwallets($password);
    if (($#{$wlist}==0) && ($wlist->[0]{error})) {
        error("FCC::wallet::savewallet - Adding wallet with wrong password")
    }
    push @{$wlist},$wallet;
    savewallets($wlist,$password)
}

# Save wallets to file
sub savewallets {
    my ($wlist,$password) = @_;
    # will overwrite password, be careful
    my $enc="";
    if ($password) {
        my $seed=""; for (my $i=0;$i<8;$i++) { $seed.=hexchar(int rand(16)) }
        $enc=$seed.securehash($seed.$COIN.$password)
    }
    my $wcl=[];
    foreach my $w (@$wlist) {
        my $wallet = {};
        if ($w->{name}) { $wallet->{name} = $w->{name} }
        if ($w->{wallet}) { $wallet->{wallet} = $w->{wallet} }
        if ($w->{pubkey}) { $wallet->{pubkey} = $password ? fccencode(hexoct($w->{pubkey}),
$password) : $w->{pubkey} }
        if ($w->{privkey}) { $wallet->{privkey} = $password ? fccencode(hexoct($w->{privkey}),
$password) : $w->{privkey} }
        # Handle types
        for my $type ('private','public','contact') {
            if (ref($w->{$type})) {
                $wallet->{$type}=$w->{$type};
                # Handle Debit
                if (ref($wallet->{$type}{Debit})) {
                    # Handle Debit Coin
                    for my $c (keys %{$wallet->{$type}{Debit}}) {
                        # Handle Debit Coin Wallets
                        for my $d (@{$wallet->{$type}{Debit}{$c}}) {
                            if ($d->{pubkey}) { $d->{pubkey}=$password ? fccencode(hexoct($d-
>{pubkey}),$password) : $d->{pubkey} }
                            if ($d->{privkey}) { $d->{privkey}=$password ? fccencode(hexoct($d-
>{privkey}),$password) : $d->{privkey} }
                        }
                    }
                }
            }
        }
        push @$wcl,$wallet
    }
    gfio::create("$WALLETDIR/wallet$FCCEXT",toJSON({ encoded => $enc, version => '2.1', wlist =>
$wcl }))
}

# Load wallet from file
sub loadwallet {
    my ($wkey,$password) = @_;
    if (defined $wkey) { $wkey=uc($wkey) }
    my $wlist=loadwallets($password);
    if (($#{$wlist}==0) && ($wlist->[0]{error})) { return $wlist->[0] }
    if (validwallet($wkey)) {
        foreach my $wallet (@$wlist) {
            if ($wallet->{wallet} eq $wkey) { return $wallet }
        }
    } elsif (!$wkey && ($#{$wlist}>=0)) {
```

```
        my $wallet=$wlist->[0]; return $wallet
    }
    return undef
}

# Validate wallet password
sub validwalletpassword {
    my ($password) = @_;
    if (-e "$WALLETDIR/wallet$FCCEXT") {
        my $winfo=dec_json(gfio::content("$WALLETDIR/wallet$FCCEXT"));
        if (ref($winfo) eq 'HASH') {
            if ($winfo->{encoded}) {
                my $seed=substr($winfo->{encoded},0,8);
                my $hash=substr($winfo->{encoded},8);
                my $phash=securehash($seed.$COIN.$password);
                return ($phash eq $hash)
            }
        }
    }
    return 1
}

# EOF FCC::wallet (C) 2018 Domero
```

Explanation of FCC::wallet.pm

Constants and Variables:

- **@EXPORT**: `$WALLETEXISTS`, `$WALLETDIR`, and various functions related to wallet management (`publichash`, `validatehash`, `createwalletaddress`, `walletexists`, `walletisencoded`, `validwalletpassword`, `newwallet`, `validwallet`, `loadwallet`, `loadwallets`, `savewallet`, `savewallets`).

Functions:

- **createtable**: Creates an XOR table for checksum calculation used in wallet address validation.
- **findwallet**: Searches for a wallet file (`wallet$FCCEXT`) in the current and parent directories.
- **publichash**: Retrieves the public hash key from a wallet object or string.
- **validatehash**: Validates a wallet address against a given public key.
- **createwalletaddress**: Generates a wallet address from a public key.
- **newwallet**: Generates a new wallet with a name, public/private key pair, and wallet address.
- **validwallet**: Validates the format of a wallet address.
- **walletexists**: Checks if a wallet file exists.
- **walletisencoded**: Checks if the wallet file is encoded (encrypted).
- **loadwallets**: Loads wallets from a file, optionally decrypting with a password.
- **savewallet**: Saves a wallet object to a file, appending it to existing wallets.
- **savewallets**: Saves a list of wallet objects to a file, overwriting existing wallets.
- **loadwallet**: Loads a specific wallet from a file based on its wallet address or returns the first wallet in the list.
- **validwalletpassword**: Validates the password for an encoded (encrypted) wallet file.

These functions collectively manage the creation, validation, loading, saving, and encryption of wallets (`FCC::wallet`) in the context of the specified application or system. Adjustments can be made based on specific application requirements or additional functionalities needed.

Summary

The **FCC::wallet.pm** package is essential within the FCC ecosystem, providing core functionalities for creating, managing, and securing cryptocurrency wallets. Leveraging cryptographic libraries and Perl’s capabilities, it ensures robustness in wallet address generation, validation, and transaction management. This module enables users to securely store and transact FCC, maintaining integrity and confidentiality through encryption and secure hash algorithms.

Introduction to FCC::leaf.pm Package

The **FCC::leaf.pm** package, introduced in version 2.01 in 2019 by Chaosje and Domero, is a crucial component of the FCC (Financial Cryptocurrency) project. It provides functionality for managing "leaves," which are less strict nodes that participate in the FCC network and perform various cryptocurrency-related tasks.

Constants and Configuration

- **$DEBUG**: Flag for enabling debug messages.
- **$LOOPWAIT**: Delay in milliseconds to release CPU during loops.
- **$FCCFUNCTION**: Defines the function type ('leaf' or 'miner').
- **$CALLER**: Reference to the caller function.
- **$LEAVES**: Array reference storing active leaves.
- **$LEAFID**: Incremental identifier for each leaf instance.
- **$VERS**: Version of the FCC system.
- **$TRANSID**: Transaction identifier, incremented for each transaction.

Dependencies

- **JSON**: Module for handling JSON data.
- **gerr**, **gfio**, **gclient**, **gserv**: Custom modules providing error handling, file operations, client-server communication, and server-side functionality.
- **Digest::SHA**, **IO::Socket::INET**, **Time::HiRes**: Perl modules for cryptographic hashing, network socket handling, and high-resolution time operations.
- **FCC::global 2.3.1**: External module providing global functionalities.
- **FCC::wallet 2.1.4**: External module providing wallet management functions.
- **FCC::fcc 1.2.6**: External module for FCC-specific functionalities.

Initialization and Setup

- **startleaf**: Initiates a new leaf instance, connecting it to a specified host and port, and registering a caller function.
- **handle_leaf**: Handles incoming commands and data from the FCC network, directing them to appropriate handler functions.

System Functions

- **outnode**: Prepares data to send to the FCC network.
- **leafloop**: Manages the main loop for passive mode leaves, ensuring buffered data is sent.

- **closeleaf**: Closes a leaf instance, notifying the caller function and terminating the connection.

Callable Functions

- **balance**: Requests the balance of a specified wallet from the FCC network.
- **transfer**: Initiates a new transaction to transfer funds between wallets.
- **sign**: Sends a signature for a specified transaction to the FCC network.
- **history**, **solution**, **ledgerinfo**, **getledgerdata**: Placeholder functions for fetching transaction history, solutions, ledger information, and ledger data.

Handle Input Functions

- **handleinput**: Processes incoming data from the FCC network, decoding JSON and invoking appropriate callback functions based on command types.
- **c_*** functions (e.g., **c_error**, **c_hello**, **c_quit**): Callback functions handling specific commands received from the FCC network, invoking the caller function with relevant data.

The FCC::leaf.pm Package

```
perl

package FCC::leaf;

################################################################
#                                                              #
#     FCC Leaf v2.01                                           #
#                                                              #
#    (C) 2019 Chaosje, Domero                                  #
#    Leaves are less strict, the node will check all           #
#                                                              #
################################################################

use strict;
no strict 'refs';  # Disable strict refs to allow symbolic references
use warnings;
use Exporter;       # Import Exporter module for exporting functions
use vars qw($VERSION @ISA @EXPORT @EXPORT_OK);

$VERSION     = '2.1.2';  # Package version
@ISA         = qw(Exporter gclient);  # Inherit from Exporter and gclient
@EXPORT      = qw();  # Export nothing by default
@EXPORT_OK   = qw(startleaf leafloop outnode closeleaf balance solution sign transfer);  # Export
specific functions

use JSON;          # JSON handling
use Digest::SHA qw(sha256_hex sha512_hex);  # SHA hashing functions
use IO::Socket::INET;  # Network socket operations
use Time::HiRes qw(gettimeofday usleep);  # High-resolution time operations

use gerr qw(error);  # Error handling from gerr module
use gfio 1.10;     # File I/O from gfio module (version 1.10)
use gclient 7.7.3;  # Client-server communication (version 7.7.3)
use gserv 4.3.2;   # Server-side functionalities (version 4.3.2)

use FCC::global 2.3.1;  # Global functionalities (version 2.3.1)
use FCC::wallet 2.1.4 qw(validwallet);  # Wallet operations (version 2.1.4)

my $DEBUG = 0;  # Debug mode flag
my $LOOPWAIT = 1000;  # Loop wait time in milliseconds
my $FCCFUNCTION='leaf';  # Default FCC function type
my $CALLER;  # Caller function reference
my $LEAVES=[];  # Array reference to store leaf instances
my $LEAFID=0;  # Leaf ID counter
my $VERS=join('.',substr($FCCVERSION,0,2)>>0,substr($FCCVERSION,2,2));  # Version string based on
FCCVERSION
```

```
my $TRANSID=((int(rand(1000000))+10000)<<20)+int(rand(1000000));  # Transaction ID generation

1;  # End of package initialization

# Function to start a new leaf instance
sub startleaf {
    my ($host,$port,$caller,$active,$miner) = @_;

    # Validate caller function
    if (!defined $caller || (ref($caller) ne 'CODE')) {
        error "Caller-function missing in FCC::leaf::start";
    }

    $CALLER = $caller;  # Set caller function

    # Set default host and port if not provided
    if (!$host) { $host='127.0.0.1' }
    if (!$port) { $port=7050 }

    $FCCFUNCTION='leaf';  # Set FCC function type
    if ($miner) { $FCCFUNCTION='miner' }  # Adjust FCC function type if miner

    $LEAFID++;  # Increment leaf ID

    # Establish WebSocket connection
    my $leaf = gclient::websocket($host, $port, $active, \&handle_leaf, undef, $::LEAF_TIMEOUT ||
5);

    # Handle connection errors
    if ($leaf->{error}) {
        print STDOUT "\nError connecting $FCCFUNCTION: $leaf->{error}\n\n";
        return $leaf;
    }

    $leaf->{connected} = 0;  # Set connection state to disconnected
    $leaf->{leafcaller} = $caller;  # Set caller function for the leaf
    $leaf->{passive} = 1;  # Set leaf mode to passive
    $leaf->{leafid} = $LEAFID;  # Set leaf ID
    $leaf->{outbuffer} = [];  # Initialize outgoing buffer

    bless($leaf);  # Bless the leaf object
    push @$LEAVES, $leaf;  # Push leaf object into LEAVES array

    return $leaf;  # Return the created leaf object
}

# Handle incoming messages from the leaf node
sub handle_leaf {
    my ($leaf, $command, $data) = @_;

    $data ||= "";  # Set $data to empty string if undefined

    # Initialization command handler
    if ($command eq 'init') {
        if (!$leaf->{passive}) {
            # Reset leaf state for active mode to allow multiple processes
            $leaf->{connected} = 0;
            $leaf->{leafcaller} = $CALLER;
            $leaf->{fccfunction} = $FCCFUNCTION;
            $leaf->{leafid} = $LEAFID;
            $leaf->{outbuffer} = [];
        } else {
            return;  # Exit if in passive mode
        }
    }

    my $func = $leaf->{leafcaller} || $CALLER;  # Set function to caller or default caller

    # Debug message output
    if ($DEBUG && ($command ne 'loop')) {
        print STDOUT " < [LEAF]: $command - $data\n";
    }

    # Command handlers
    if ($command eq 'loop') {
        # No action needed for 'loop' command
    } elsif ($command eq 'input') {
```

```
        handleinput($leaf, $data);  # Handle input command
    } elsif ($command eq 'error') {
        gclient::wsquit($leaf);  # Quit WebSocket connection
        print STDOUT "Leaf exited with error: $data\n\n";  # Print error message
        &$func($leaf, 'disconnect', { error => $data });  # Call disconnect callback
    } elsif ($command eq 'quit') {
        print STDOUT "Lost connection to node: $data\n\n";  # Print disconnect message
        &$func($leaf, 'disconnect', { error => $data });  # Call disconnect callback
    } elsif ($command eq 'close') {
        print STDOUT "Lost connection to node: $data\n\n";  # Print close message
        &$func($leaf, 'disconnect', { error => $data });  # Call disconnect callback
    } elsif ($command eq 'connect') {
        my ($tm, $ip) = split(/ /, $data);  # Split data into time and IP
        $leaf->{connected} = 1;  # Set connected state to true
        if ($DEBUG) {
            print STDOUT prtm() . "Connected as $leaf->{fccfunction} v$VERS at $leaf->{localip} to
$ip\n";
        }
    }
}

################## SYSTEM FUNCTIONS ##########################

# Function to push data to the outgoing buffer of a leaf instance
sub outnode {
    my ($leaf, $k) = @_;
    if (ref($k) ne 'HASH') {
        error "Not a hash-reference given in FCC::leaf::outnode";  # Error if $k is not a hash
reference
    }
    push @{$leaf->{outbuffer}}, $k;  # Push data to outbuffer
}

# Function to manage the main loop for passive mode leaves
sub leafloop {
    # Iterate through each leaf in LEAVES array
    foreach my $leaf (@$LEAVES) {
        if ($leaf->{connected}) {  # Check if leaf is connected
            if ( @{$leaf->{outbuffer}} ) {  # Check if outbuffer is not empty
                my $json = JSON->new->allow_nonref;  # Create JSON object
                my $data = shift @{$leaf->{outbuffer}};  # Dequeue data from outbuffer
                gclient::wsout($leaf, $json->encode($data));  # Encode and send data over WebSocket
            }
        }
        $leaf->takeloop();  # Execute takeloop method on leaf
    }
}

# Function to terminate a leaf instance
sub closeleaf {
    my ($leaf, $msg) = @_;
    if (!$msg) { $msg = 'Closed' }  # Set default message if not provided
    $leaf->{leafcaller}->($leaf, 'terminated', { message => $msg });  # Call terminated callback
    if ($leaf->{connected}) {
        $leaf->wsquit($msg);  # Quit WebSocket connection
    } else {
        $leaf->quit($msg);  # Quit connection
    }
}

################## CALLABLE FUNCTIONS ####################################

# Function to check balance of a wallet
sub balance {
    my ($leaf, $wallet) = @_;
    if (ref($wallet)) { $wallet = $wallet->{wallet} }  # Get wallet ID if wallet is a reference
    outnode($leaf, { command => 'balance', wallet => $wallet });  # Push balance command to
outbuffer
}

# Function to initiate a transfer transaction
sub transfer {
    my ($leaf, $pubkey, $changewallet, $tolist) = @_;
    $TRANSID++;  # Increment transaction ID
    outnode($leaf, { command => 'newtransaction', transid => $TRANSID, pubkey => $pubkey, to =>
$tolist, changewallet => $changewallet });  # Push transaction data to outbuffer
}
```

```
# Function to sign a transaction
sub sign {
    my ($leaf, $transid, $signature) = @_;
    outnode($leaf, { command => 'signtransaction', transid => $transid, signature => $signature });
# Push sign transaction command to outbuffer
}

# Placeholder function for transaction history
sub history {
    my ($leaf, $wallet) = @_;
    # Placeholder - implement as needed
}

# Placeholder function for solution handling
sub solution {
    my ($leaf, $wallet, $solhash) = @_;
    outnode($leaf, { command => 'solution', wallet => $wallet, solhash => $solhash });  # Push
solution command to outbuffer
}

# Function to request ledger information
sub ledgerinfo {
    my ($leaf) = @_;
    outnode($leaf, { command => 'ledgerinfo' });  # Push ledgerinfo command to outbuffer
}

# Function to request ledger data
sub getledgerdata {
    my ($leaf, $pos, $length, $final) = @_;
    if (!$final) { $final = 0 }  # Set default value for $final if not provided
    outnode($leaf, { command => 'reqledger', pos => $pos, length => $length, final => $final });  #
Push reqledger command to outbuffer
}

################## HANDLE INPUT ###########################################

# Function to handle incoming data from the FCC network
sub handleinput {
    my ($leaf, $data) = @_;
    my $json = JSON->new->allow_nonref;  # Create JSON object
    my $k = $json->decode($data);  # Decode JSON data

    my $cmd = $k->{command};  # Get command from decoded data

    if ($k->{error}) {
        # Handle error response
        $leaf->{leafcaller}->($leaf, 'error', { command => 'error', message => $cmd, error => $k-
>{error}, available => $k->{available}, spendable => $k->{spendable} });
        return;
    }

    my $proc = "c_$cmd";  # Create function name based on command

    if (defined &$proc) {
        # Call specific command handler if defined
        &$proc($leaf, $k);
    } else {
        # Handle illegal command
        if (defined $::ILLEGAL_CALLBACK && ref($::ILLEGAL_CALLBACK) eq 'CODE') {
            &$::ILLEGAL_CALLBACK($data);  # Call illegal callback function
        } else {
            print STDOUT "Illegal command sent to leaf: [$cmd]\n";  # Print illegal command message
        }
    }
}

# Callback function for 'error' command
sub c_error {
    my ($leaf, $k) = @_;
    print STDOUT "Error: $k->{message}\n";  # Print error message
    outnode($leaf, { command => 'quit' });  # Push quit command to outbuffer
    $leaf->{leafcaller}->($leaf, 'error', $k);  # Call error callback
    $leaf->quit();  # Quit connection
}

# Callback function for 'hello' command
```

```
sub c_hello {
    my ($leaf, $k) = @_;
    outnode($leaf, { command => 'identify', type => $leaf->{fccfunction}, version => $FCCVERSION });
# Push identify command to outbuffer
    $leaf->{leafcaller}->($leaf, 'response', { node => "$k->{host}:$k->{port}", version => $k-
>{version} });  # Call response callback
}

# Callback function for 'quit' command
sub c_quit {
    my ($leaf, $k) = @_;
    $leaf->{leafcaller}->($leaf, 'disconnect', $k);  # Call disconnect callback
    $leaf->quit();  # Quit connection
}

# Callback function for 'balance' command
sub c_balance {
    my ($leaf, $k) = @_;
    $leaf->{leafcaller}->($leaf, 'balance', $k);  # Call balance callback
}

# Callback function for 'newtransaction' command
sub c_newtransaction {
    my ($leaf, $k) = @_;
    if ($k->{error}) {
        $leaf->{leafcaller}->($leaf, 'transstatus', $k);  # Call transstatus callback if error
    } else {
        $leaf->{leafcaller}->($leaf, 'sign', { data => $k->{sign}, transid => $k->{transid} });  #
Call sign callback with data
    }
}

# Callback function for 'signtransaction' command
sub c_signtransaction {
    my ($leaf, $k) = @_;
    $leaf->{leafcaller}->($leaf, 'transstatus', $k);  # Call transstatus callback
}

# Callback function for 'processed' command
sub c_processed {
    my ($leaf, $k) = @_;
    if ($k->{error}) {
        $leaf->{leafcaller}->($leaf, 'transstatus', $k);  # Call transstatus callback if error
    } else {
        $k->{status} = 'success';  # Set status to success
        $leaf->{leafcaller}->($leaf, 'transstatus', $k);  # Call transstatus callback
    }
}

# Placeholder callback function for 'history' command
sub c_history {
    my ($leaf, $k) = @_;
    my $func = $leaf->{leafcaller};
    # Implement as needed
}

# Callback function for 'mine' command
sub c_mine {
    my ($leaf, $k) = @_;
    $leaf->{leafcaller}->($leaf, 'mine', $k);  # Call mine callback
}

# Callback function for 'solution' command
sub c_solution {
    my ($leaf, $k) = @_;
    $leaf->{leafcaller}->($leaf, 'solution', $k);  # Call solution callback
}

# Callback function for 'wrong' command
sub c_wrong {
    my ($leaf, $k) = @_;
    if (defined $::WRONG_CALLBACK && ref($::WRONG_CALLBACK) eq 'CODE') {
        $::WRONG_CALLBACK->($k);  # Call WRONG_CALLBACK if defined
    } else {
        $leaf->{leafcaller}->($leaf, 'solerror', $k);  # Call solerror callback
    }
}
```

```
# Callback function for 'ledgerresponse' command
sub c_ledgerresponse {
    my ($leaf, $k) = @_;
    $leaf->{leafcaller}->($leaf, 'ledgerinfo', $k);  # Call ledgerinfo callback
}

# Callback function for 'ledgerdata' command
sub c_ledgerdata {
    my ($leaf, $k) = @_;
    $leaf->{leafcaller}->($leaf, 'ledgerdata', $k);  # Call ledgerdata callback
}

# EOF leaf.pm (C) 2018 Chaosje, Domero
```

Explanation of FCC::leaf.pm

Constants and Variables:

- **$LOOPWAIT**: Interval (in milliseconds) for the main loop to wait before processing the next iteration.
- **$FCCFUNCTION**: Indicates the current function of the leaf (`leaf` or `miner`).
- **$CALLER**: Stores the callback function for handling leaf events.
- **$LEAVES**: Array reference storing active leaf instances.
- **$LEAFID**: Unique identifier assigned to each leaf instance.
- **$VERS**: Version of the FCC system.
- **$TRANSID**: Transaction ID for identifying individual transactions.

Functions:

- **startleaf**: Initializes a new leaf instance and establishes a WebSocket connection to a specified host and port. It sets up callback functions for handling network events.
- **handle_leaf**: Handles incoming messages and commands from the FCC network. It dispatches commands to appropriate handler functions based on the received data.
- **outnode**: Adds data to the outgoing buffer of a leaf instance, preparing it for transmission over the network via WebSocket.
- **leafloop**: Manages the main loop for passive mode leaves. It ensures that outgoing data is transmitted and periodic operations are performed.
- **closeleaf**: Terminates a leaf instance, performing cleanup tasks and notifying callback functions about the termination event.
- **balance**: Requests balance information for a specified wallet from the FCC network.
- **transfer**: Initiates a new transaction, specifying the recipient's public key, change wallet, and list of recipients.
- **sign**: Signs a transaction using the leaf's private key, ensuring transaction authenticity and validity.
- **solution**: Handles solution-related tasks, such as verifying solutions to cryptographic puzzles (mining) within the FCC network.

- **ledgerinfo**: Requests general ledger information from the network, providing insights into network-wide transaction data.
- **getledgerdata**: Retrieves specific ledger data based on position, length, and finality parameters, enabling detailed analysis of transaction history and network status.

Summary of FCC::leaf.pm

The **FCC::leaf.pm** Perl package facilitates the management of less strict nodes (leaves) within the FCC cryptocurrency network. It includes functionalities for network communication, transaction handling, and data retrieval. Here are the key aspects:

- **Network Communication**: Utilizes WebSocket connections for communication between nodes, facilitating real-time data exchange.
- **Transaction Management**: Provides functions for initiating transactions (`transfer`), signing transactions (`sign`), checking balances (`balance`), and handling transaction histories (`history`).
- **Data Handling**: Manages ledger information retrieval (`ledgerinfo`), fetches ledger data (`getledgerdata`), and processes solutions (`solution`) within the network.
- **Error Handling**: Implements robust error handling and debugging features (`handleinput`) to manage and log errors encountered during node operations.
- **Callback Mechanism**: Utilizes callback functions (`c_*`) to handle specific commands received from the FCC network, ensuring appropriate responses to various network events.
- **Integration with Modules**: Integrates with various Perl modules (`JSON`, `gerr`, `gfio`, `Digest::SHA`, etc.) to enhance functionality and ensure secure and efficient operations within the FCC network ecosystem.

Overall, **FCC::leaf.pm** serves as a crucial component for managing node interactions within the FCC cryptocurrency network, supporting reliable and secure transaction processing and network communication.

11.3 – Common Errors and Troubleshooting

Throughout this manual, common errors and their solutions have been described to assist readers in resolving issues promptly. However, for less common errors or situations not covered here, we recommend reaching out to our support team. Our dedicated support staff is available to provide personalized assistance and address specific issues that may arise.

For the best possible resolution and to ensure smooth operation of your system, please contact our support team. We are committed to helping you overcome any challenges you encounter and ensuring your experience with our product is as smooth and effective as possible.

11.4 – Contact Information for Support

For any inquiries, issues, or suggestions related to Factorial Coin (FCC), please reach out to our support team via email at factorialcoin@gmail.com. Our dedicated support staff is available to assist you promptly and ensure your experience with Factorial Coin is smooth and productive.

Feedback and Updates

We welcome your feedback on the manual's content and are open to suggestions for improvements or corrections. If you discover any errors, ambiguities, or areas where further clarification is needed within this manual, please do not hesitate to contact us. Your input helps us continually refine and enhance the quality of our documentation.

Additional Assistance

If you require additional information beyond what is covered in this manual, or if you encounter uncommon errors or issues not addressed here, our support team is ready to provide personalized assistance. We strive to ensure that all users have access to the resources they need for a successful experience with Factorial Coin.

Contact Email: factorialcoin@gmail.com