Operating Systems Project Documentation

Project 1

Team members :

- Behraz Fereshteh Saniee
- 1401012268053
- Rozhina Latifi
- 1401012268001

Bank Module Documentation

This document provides an overview of the `bank.cpp` file, which implements a banking system for handling cryptocurrency transactions. It includes functionalities for managing clients, exchanges, and cryptocurrency operations.

Global Variables

`unordered_map clients`

A map that stores client information indexed by their port number.

`unordered_map exchanges`

A map that keeps track of exchanges indexed by their port number.

`ThreadSafeBuffer input_buffer`

A thread-safe buffer for storing incoming messages from clients and exchanges.

`constexpr int max_increase_amount`

The maximum amount that a client can increase their balance in a single request.

`constexpr int page_size`

The number of history records displayed per page when a client requests their transaction history.

`auto timer`

A timer used to control periodic updates to cryptocurrency prices.

`vector history`

A vector that stores a history of transactions and operations for logging purposes.

`unordered_map cryptocurrencies`

A map that contains information about different cryptocurrencies.

Functions

`void bank_reader()`

This function is responsible for reading messages from clients and exchanges. It sets up a UDP socket, binds to a specific port, and continuously listens for incoming messages. When a message is received, it adds the message to the input buffer for processing.

Example: When a client sends a transaction request, the message is received and stored for further processing.

`void request_handler()`

This function handles requests from the input buffer. It processes various commands such as initializing clients and exchanges, viewing balances, and handling cryptocurrency transactions. It also periodically updates cryptocurrency prices based on the clients' holdings.

Example: If a client wants to check their balance, the request handler retrieves the balance from the client's information and sends it back.

Entry Point

`int main()`

The `main` function initiates the bank module by starting two threads: one for reading messages and another for handling requests. It ensures that the program runs concurrently, allowing for efficient processing of multiple transactions.

Example: The program will listen for client requests while simultaneously processing incoming messages, ensuring a smooth experience for users.

The `bank.cpp` file serves as the core of the banking system, managing client interactions, cryptocurrency transactions, and maintaining a history of operations.

Client Application Documentation

This documentation provides a friendly overview of the client application which interacts with a cryptocurrency server. The client allows users to perform various operations related to cryptocurrency transactions.

Overview of the Code File

The `client.cpp` file implements a client that communicates with a cryptocurrency server using UDP sockets. Users can query cryptocurrency prices, buy or sell cryptocurrencies, view their wallet balance, and check transaction history.

Global Variables

`string menu[]`

An array of strings that represents the options available to the user in the main menu. Each option corresponds to a specific action related to cryptocurrency transactions.

`int assigned_port`

This variable holds the port number assigned to the client by the operating system upon socket binding.

`string name`

This variable stores the name of the user, which is required for registration with the server.

Functions

`void get_cryptocurrency_price(int c_socket_fd, int sock_fd, sockaddr_in sock_in)`

This function prompts the user to enter a cryptocurrency name, constructs a message to request its price from the server, and sends this message. It then waits for a response from the server and displays it.

```
get_cryptocurrency_price(client_socket, server_socket, server_address);
```

`void get_exchanges_list(int c_socket_fd, int sock_fd, sockaddr_in sock_in)`

This function sends a request to the server to obtain a list of available exchanges. It waits for the server response and displays the list to the user.

```
get_exchanges_list(client_socket, bank_socket, bank_address);
```

`void buy_cryptocurrency(int c_socket_fd, int sock_fd, sockaddr_in sock_in)`

This function allows the user to buy a specified amount of a cryptocurrency. It constructs a request message and sends it to the server, then displays the server's response.

```
buy_cryptocurrency(client_socket, exchange_socket, exchange_address);
```

`void sell_cryptocurrency(int c_socket_fd, int sock_fd, sockaddr_in sock_in)`

Similar to the buy function, this one handles selling a specified amount of a cryptocurrency. It constructs the necessary request

and processes the response from the server.

```
sell_cryptocurrency(client_socket, exchange_socket, exchange_address);
```

`void view_wallet_balance(int c_socket_fd, int sock_fd, sockaddr_in sock_in)`

This function sends a request to retrieve the user's wallet balance from the server and displays the response.

```
view_wallet_balance(client_socket, bank_socket, bank_address);
```

`void increase_wallet_balance(int c_socket_fd, int sock_fd, sockaddr_in sock_in)`

Allows the user to request an increase in their wallet balance. The function sends the request and waits for the server's response.

```
increase_wallet_balance(client_socket, bank_socket, bank_address);
```

`void view_transaction_history(int c_socket_fd, int sock_fd, sockaddr_in sock_in)`

This function allows the user to view their transaction history by sending a request to the server and displaying the returned data.

```
view_transaction_history(client_socket, bank_socket, bank_address);
```

Main Function

The `main` function initializes the client application. It handles user input to navigate through the menu options and manage socket connections with both the bank and exchange servers. Users are prompted to enter their name and select actions from the menu.

Example of User Interaction:

```
Enter your name: Alice
Client is listening on assigned port: 12345
0: Get cryptocurrency price
1: Get exchanges list
2: Buy cryptocurrency
3: Sell cryptocurrency
4: View wallet balance
5: Requesting a balance increase
6: View transaction history
7: Exit
```

Exchange Module Documentation

This document provides a comprehensive overview of the Exchange module found in the `os_project/exchange/exchange.cpp` file. This module facilitates the management of cryptocurrency transactions, enabling clients to buy and sell cryptocurrencies efficiently.

Overview

The Exchange module handles client requests related to cryptocurrency trading. It listens for incoming UDP messages, processes various transaction requests, and communicates with a bank server to manage cryptocurrency inventories and balances.

Global Variables

- **string name;** - Stores the name of the exchange.
- **int assigned_port;** - Holds the port number assigned to the exchange.
- **ThreadSafeBuffer input_buffer;** - Buffer for safely storing incoming client requests.
- **Safe safe;** - An object that holds the current state of available cryptocurrencies and their prices.
- **constexpr double init_price;** - The initial price for new cryptocurrencies.

Functions

void exchange_reader()

This function sets up a UDP socket to listen for incoming messages from clients and the bank server. It handles client registrations and processes incoming requests.

```
This function is executed in its own thread and runs continuously.
```

void request_handler()

This function processes requests from the input buffer. It handles various commands such as fetching prices, adding new cryptocurrencies, buying, and selling cryptocurrencies.

```
This function runs in a separate thread to ensure it can handle requests concurrently.
```

void new_crypto_handler()

This function allows the user to add new cryptocurrencies by entering their names. It sends the registration request to the bank server.

```
This function runs in its own thread, prompting the user for new cryptocurrency names.
```

int main()

This is the main function that initializes the exchange. It starts three threads: one for reading messages, one for handling requests, and one for adding new cryptocurrencies.

requests, and one for adding new cryptocurrencies.

```
The program begins execution here, allowing the user to input the exchange name.
```

Documentation for `constant_and_types.h`

This header file is part of an operating system project and defines several constants and data structures used throughout the application. The file contains essential settings and types that help manage user wallets, safe items, and client and exchange information.

Constants

The following constants are defined in this file:

- **BUFFER_SIZE**: A constant integer set to `4096`. This value typically represents the size of data buffers used in the application.
- **BANK_PORT**: A constant integer set to `9999`. This value is likely used to specify the port for bank-related communications.
- **server_ip**: A constant string initialized to `"127.0.0.1"`, which represents the localhost IP address where the server is expected to run.

Structures

This file defines several key data structures that are used to manage financial transactions and user data:

Wallet

This structure represents a user's wallet, storing their balance and the cryptocurrencies they own.

- **balance**: A double that holds the current balance of the wallet.
- **cryptocurrencies**: An unordered map that links cryptocurrency names (as strings) to their respective quantities (as integers).
- **pars_string()**: A method that converts the wallet's balance and its cryptocurrencies into a formatted string for easy representation.

Example usage: If a wallet has a balance of 100.0 and contains 2 Bitcoin and 5 Ethereum, calling `pars_string()` would return a string like: `"100.0 Bitcoin 2 Ethereum 5 "`.

SafeItem

This structure represents an item in a safe, which can be bought or sold. It includes the following fields:

- **buying**: A boolean indicating whether the item is currently being bought.
- **count**: An integer that shows how many units of the cryptocurrency are available.
- **init_count**: An integer that indicates the initial count of the item when it was created.
- **cryptocurrency**: A string that names the cryptocurrency associated with this item.
- **price**: A double representing the price of the cryptocurrency.
- **state**: A string that indicates the current state of the item (default is `"preorder"`).
- **creation_time**: A time point that records when the item was created.

Safe

This structure holds information about a safe that contains various cryptocurrencies.

- **balance**: A double representing the total balance in the safe.

- **balance**: A double representing the total balance in the safe.
- **cryptocurrencies**: An unordered map that connects cryptocurrency names to their respective SafeItem objects.

ClientInfo

This structure contains information about a client using the system, including:

- **name**: A string representing the client's name.
- **port**: An integer indicating the communication port for this client.
- **wallet**: A Wallet object that stores the client's financial information.

ExchangeInfo

This structure provides details about an exchange, which includes:

- **name**: A string that denotes the name of the exchange.
- **port**: An integer indicating the port for the exchange's operations.

Documentation for HashFunction.h

This header file defines a simple hashing function that can be used to generate a hashed representation of a given message.

File Overview

The `HashFunction.h` file contains the declaration of a simple hashing function called `simpleHash`. This function takes a string input (the message) and returns a hashed string. The hashing is enhanced by adding a secret key to the input.

Constants

SECRET

Type: `const string`

Value: "SecretKey:)"

This constant string is used as a secret key that gets appended to the input message before hashing. It adds an extra layer of complexity to the hash function, making it less predictable.

Function: simpleHash

Declaration: `inline string simpleHash(const string& message);`

This function generates a hashed string from the provided message. Here's how it works:

- The function begins by defining a constant `size` which determines the desired length of the output hash string.
- A variable `hash` is initialized with a starting value.
- The input message is concatenated with the `SECRET` key.
- It then iterates over each character of the modified input, updating the hash value based on a simple algorithm.
- The resulting hash is converted into a string.
- If the hash string exceeds the defined size, it is truncated. If it's shorter, it's padded with the character 'P' until it reaches the desired length.

Return Value: The function returns a string that represents the hashed output of the input message.

Example Usage:

Here is a simple example of how to use the `simpleHash` function:

```
string message = "Hello, World!";
string hashedMessage = simpleHash(message);
```

In this example, the output `hashedMessage` will contain the hashed version of "Hello, World!" combined with the `SECRET` key.

ThreadSafeBuffer Documentation

This documentation provides a comprehensive overview of the `ThreadSafeBuffer` class, which is designed to handle a buffer in a thread-safe manner. It allows multiple threads to add and remove items safely, ensuring that data integrity is maintained even in concurrent environments.

Overview

The `ThreadSafeBuffer` class is a template class that implements a thread-safe queue using mutexes and condition variables. It provides methods to add and remove items from the buffer while preventing data races. The buffer can also be configured to log actions for better debugging and monitoring.

Class Definition

`template`

This line defines a template class, allowing `ThreadSafeBuffer` to store any data type `T`.

Constructor

ThreadSafeBuffer(const ulli size, const string& name, bool logEn)

The constructor initializes the buffer with a specified size and name. It also sets the logging option.

Parameters:

- `size`: The maximum size of the buffer (must be greater than 0).
- `name`: A string representing the name of the buffer.
- `logEn`: A boolean that enables or disables logging.

If the specified size is less than or equal to zero, an `invalid_argument` exception is thrown.

Member Functions

void add(const T &item)

Adds an item to the buffer. If the buffer is full, it waits until space is available.

Example:

```
buffer.add(42); // Adds the integer 42 to the buffer
```

void add_drop(const T &item)

Adds an item to the buffer but drops it if the buffer is full. No waiting occurs in this case.

Example:

```
buffer.add_drop(42); // Adds the integer 42 or drops it if full
```

T remove()

Removes and returns the item from the front of the buffer. If the buffer is empty, it waits until an item is available.

Example:

```
int item = buffer.remove(); // Retrieves and removes the first item
```

T remove_no_wait()

Removes and returns the item from the front of the buffer without waiting. If the buffer is empty, it returns a default value.

Example:

```
int item = buffer.remove_no_wait(); // Retrieves and removes the first item or returns -1 if empty
```

bool is_empty()

Checks if the buffer is empty.

Example:

```
if (buffer.is_empty()) { /* Buffer is empty */ }
```

bool is_full()

Checks if the buffer is full.

Example:

```
if (buffer.is_full()) { /* Buffer is full */ }
```

ulli size()

Returns the current number of items in the buffer.

Example:

```
ulli currentSize = buffer.size(); // Gets the current size of the buffer
```

[[nodiscard]] ulli max_size() const

Returns the maximum size of the buffer.

Example:

```
ulli maxSize = buffer.max_size(); // Gets the maximum size of the buffer
```