UNIVERSITÀ DI PISA

Computer Engineering

Distributed Systems and Middleware Technologies

Distributed Cryptocurrency Exchange

Project Report

TEAM MEMBERS:
Biagio Cornacchia
Gianluca Gemini
Matteo Abaterusso

Academic Year: 2023/2024

1 Project Specifications

The application consists of a **cryptocurrency exchange**. A user can buy and sell cryptocurrencies, visualize their current market value and manage his/her wallet.

1.1 Terminology

The following are listed terms used in the report.

- **Balance**: quantity of money (euro) owned by a user
- **Asset**: cryptocurrencies available in the exchange
- **Wallet**: balance and assets owned by a user
- **Pending Order**: sell or buy order not yet filled
- **Transaction**: a couple of completed sell and buy orders

1.2 Requirements

Actors involved in the application are the following:

- Unregistered User
- User

1.2.1 Functional Requirements

The functional requirements are the following:

- Unregisterd User
  - Signup
- User
  - Login/Logout
  - Visualize a list of all assets associated with their market value
  - Choose an asset and visualize its market value history
  - Place buy and sell orders at current market value
  - Place buy and sell orders at a desired market value
  - Cancel pending orders
  - Add money and assets to his/her wallet
  - Visualize owned assets and available balance
  - Visualize pending orders and transaction history

1.2.2 Non Functional Requirements

The non functional requirements are the following:

- Strong consistency has to be guaranteed for pending orders, transactions, asset market values and user wallet.
- Manage concurrency of pending orders
- Pending orders have to be filled based on their arrival order
- Fault tolerance
- Horizontal scalability

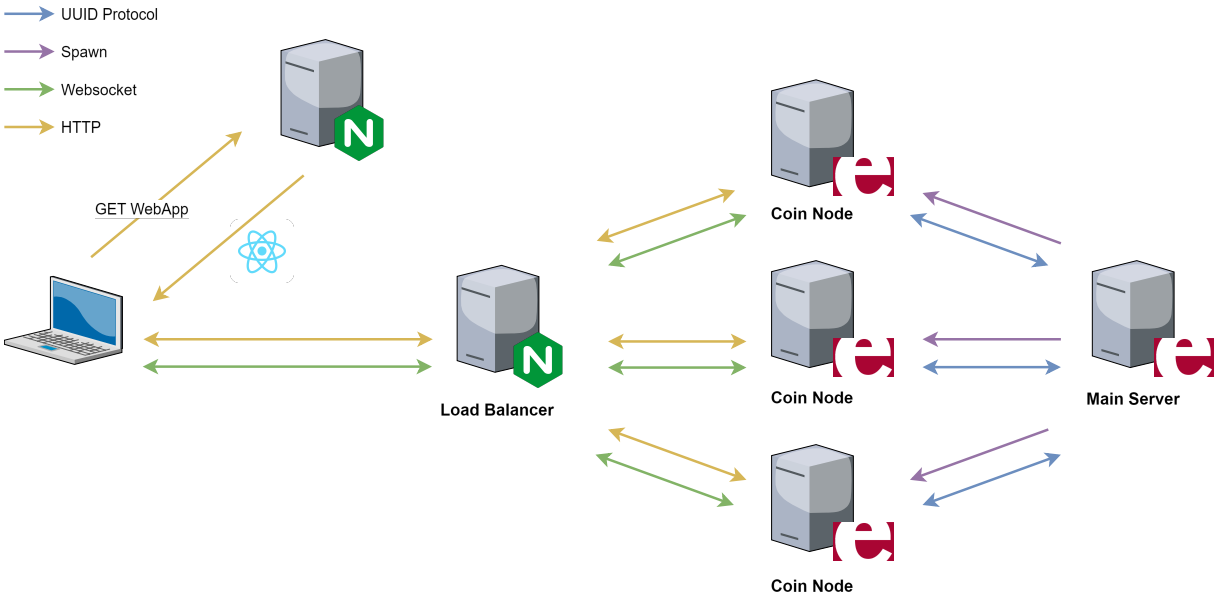
1.3 Synchronization and Communication Issues

The synchronization and communication issues that have to be managed are the following:

- Clients have to visualize the synchronized market values of assets
- Clients can place orders concurrently
- Pending orders have to be filled in fist-come first-served order
- Clients have to visualize the fulfillment of their pending orders in real time
- Clients have to visualize all transactions related to a specific asset in real time

2 Architecture

The chosen architecture is shown below.



The **Main Server** spawns the **Coin Nodes** on the available servers. The Coin Nodes handle requests coming from clients which deal with user accounts, wallets and orders management. They are **equivalent** each other, thus each Coin Node can handle any order about any asset and from any user. In order to equally distribute requests on all nodes, a *Nginx Load Balancer* has been deployed. The interaction between clients and Coin Nodes occurs through *HTTP requests* and *WebSockets*.

All data related to users, assets, pending orders, and transactions are stored in **MnesiaDB** to ensure **strong consistency**.

The frontend is a **single-page application** that **dynamically changes** based on the data received from the coin nodes. To handle this dynamicity, *ReactJS* has been chosen as the frontend development framework. Since ReactJS produces web applications in pure *HTML*, *CSS*, and *Javascript*, these are served to the client through the *Nginx web server*, which is designed specifically to serve static web apps.

2.1 Main Server

The Main Server is an **Erlang node** that implements the following operations.

- It creates the Erlang cluster by connecting nodes each other
- It generates and initializes the Mnesia database
- It handles the database replication on cluster nodes
- It spawns the Erlang process on cluster nodes
- It provides the global UUID generation protocol to cluster nodes

2.2 Coin Node

The Coin Node implements all application services requested by clients. In particular, it implements the following operations.

- It serves the signup and login request from users
- It allows to create and delete the pending orders requested by users
- It allows users to deposit and withdraw assets and money from their wallet
- It fills pending orders by transferring assets and money between seller and buyer ensuring FIFO policy
- It guarantees consistency between pending orders status and users' wallet
- It communicates the market value cryptocurrencies variation and new transactions to active clients

3 Implementation

In this section will be explained the implementation details of all elements presented in the architecture.

3.1 Main Server

At startup the Main Server creates the *Mnesia database* and loads the list of **available Erlang nodes** from the configuration. This list is iterated by the *connect_nodes* function that executes the following operations for every node.

- It calls the *net_kernel:connect_node* primitive to add the node to the cluster
- It spawns the *coin_node* process on the node
- It waits for the *ok* message reception from the spawned process to be sure that it is successfully started
- It replicates the Mnesia tables on the new connected node

Then, the Main Server generates an *OTP supervisor* that **launches** and **monitors** the *uuid_generator* process.

3.1.1 UUID Generator

A problem to address is the generation of **cluster-wide unique keys** for records related to pending orders and transactions tables. Since Mnesia doesn't provide a built-in mechanism to generate unique keys, a process to do that has been implemented. In particular, the *uuid_generator* process exploits the UUID Mnesia table to keep the status of generated UUIDs. This table is made the following fields.

- *table*: the name of the table for which id generation has been required
- *last_uuid*: next available UUID

A Coin Node requests a new UUID by contacting the *uuid_generator* via an **erlang message** structured as follows {*uuid*, *Table*, *From*}, where *Table* indicates the name of the table for which a new UUID is to be obtained, and *From* contains the PID of the remote process that sent the message. The *uuid_generator* responds with the following message {*uuid*, *UUID*} where UUID contains the *last_uuid* of the requested table which is subsequently incremented.

The *uuid_generator* lifecycle is handled by the supervisor with following configuration.

```
UUIDGenerator = #{
    id => uuid_generator,
    start => {uuid_generator, start, []},
    restart => permanent,
    shutdown => brutal_kill
}
```

In particular, the restart *permanent clause* allows to restart the process in case of crash.

3.2 Mnesia Database

About the data persistence, the Mnesia database has been chosen. Mnesia is directly integrated in Erlang and ensures **strong consistency** and **atomicity**. The tables and related Erlang records designed for the application are shown below.

The **Coin table** contains all the assets supported by the application and their market values.

```
-record(coin, {id::string(), value::float()}).
```

All data related to users are stored in the **User table**. They include the user id, password hashed in *sha-256* and the user balance (deposit).

```
-record(user, {id::string(), password::term(), deposit::float()}).
```

The assets owned by the users are stored in the **Asset table**. The record key is made by the user id and asset. In addition, there is also the owned asset quantity.

```
-record(asset_key, {user_id::string(), coin_id::string()}).
-record(asset, {asset_key, quantity::float()}).
```

A table containing **pending orders** for each asset has been created. The key is represented by an UUID generated by the Main Server. A record contains the id of the user that issued the order, the creation timestamp, the type (sell or buy), the asset and the quantity requested. The last field is the limit that is the target market value to fill the order. If the limit is 0, the pending order can be filled at any market value. This table has been created as *ordered set*, so that orders are already ordered by creation order.

```
-record(order, {uuid::integer(), timestamp::integer(), user_id::string(),
    type::string(), coin_id::string(), quantity::float(), limit::float()}).
```

A transaction in the **Transaction table** is created for each couple of filled pending orders. The key is also in this case an UUID provided by the Main Server, used to order transactions by creation order. Indeed, also this table has been created as ordered set. Every transaction is made by the creation timestamp, seller and buyer id, asset exchanged and its quantity, market value used for the transaction and the market value updated after the transaction.

```
-record(transaction, {uuid::integer(), timestamp::integer(),
    seller::string(), buyer::string(), coin_id::string(), coins::float(),
    market_value::float(), new_market_value::float()}).
```

Finally, as seen in the previous section, also a **UUID table** has been designed.

```
-record(uuid, {table::string(), last_uuid::integer()}).
```

Regarding table **replication**, the UUID table is persistent only in the Main Server, whereas all the others are replicated across every nodes. Such a policy has been chosen to avoid **remote accesses** and to guarantee the **highest fault tolerance**.

Moreover, in order to allow the concurrent execution of pending orders for different assets, a **different table** for each asset has been created.

3.3 Coin Node

The Coin Node interfaces directly with the frontend. The frontend asks for a service to the Coin Node using a **synchronous REST request**. Then, the Coin Node communicates new market values and transactions to the clients using **WebSockets**, in an **asynchronous manner**.

A library named *Cowboy* has been used to handle the REST requests and WebSockets.

The Coin Node is spawned by the Main Server. Immediately after, the supervisor is started which in turn launches and monitors two processes, the *cowboy_listener* and the *broadcast_dispatcher*. The former manages all the REST requests and WebSockets. The latter performs the broadcasting of a message to all the currently WebsSockets open.

All the REST resources and WebSockets will be presented below.

3.3.1 REST Resources

The **user** authentication and creation are managed by the following resource:

/api/authentication

The possible requests are the following:

- **POST**: the request body contains the operation type (login or signup), user id and password

The **asset** informations supported by the exchange are reachable from the following resource:

/api/coin

The possible requests are the following:

- **GET**: it doesn't require any query parameters and returns a list of all assets along with their market value

The **wallet** operation can be requested by the following resource:

/api/wallet

The possible requests are the following:

- **GET**: it requires the user id, asset (a specific one or all) and balance (a boolean used to obtain the user balance) as query parameter. It returns the owned assets quantity and possibly the user balance
- **PUT**: the request body contains the user id, the operation (balance or asset), type (deposit or withdraw) and the quantity

The **pending orders** operations can be request by the following resource:

/api/order

The possible requests are the following:

- **GET**: it requires the user id and asset as query parameters. It returns the list of pending orders for the specified asset and user
- **POST**: the request body contains the operation type (sell or buy), user id, the target asset, the quantity (asset if the operation type is a sell, euros if buy) and the limit
- **DELETE**: the request body contains the UUID of the pending order to delete and the relative asset

The **transaction** information can be retrieved using the following resource:

/api/transaction

The possible requests are the following:

- **GET**: it requires the asset and the time interval used to filter transactions. The interval is expressed in seconds and represents the max age of a transaction

3.3.2 About Sell and Buy Operations

When a new pending order is created using the */api/order* resource, the asset (or money) quantity specified by the user is **subtracted** from its wallet and **reserved** for the creation of the new order. Then, the Coin Node that received the request **spawns** a process in charge to fill as many orders as possible related to the same coin specified by the new order.

The procedure performed to fill orders is the following:

1. Obtain the oldest fillable pending order (it is the oldest order that can be coupled with another order of opposite type)
2. Perform a buy or sell, depending on the order type
3. Update the market value
4. Check if after the market value update there are orders older than the one chosen at point 1

The procedure is repeated as long as there is **at least one order** that can be filled. A buy/sell operation creates a transaction pairing the order chosen at point 1 and the oldest one available of the opposite type. During this operation, the one with the least quantity is completed and removed whereas the other one remains as pending order with the remaining quantity at the end of the operation.

The procedure presented above is **executed atomically** in order to guarantee data consistency and avoid that the same pending order can be filled by other processes.

Every time a transaction is completed, a **new market value** for the asset is computed. The function used is the following.

```
update_market_value(MarketValue, Type, CoinId, Quantity) ->
    {ok, Orders} = get_orders_by_coin(CoinId),
    {TotalBuy, TotalSell} = count_orders(Orders, 0, 0),
    PendingWeigth = (convert_currency_to_asset(MarketValue, TotalBuy) -
        TotalSell) * 0.0001,
```

```
        if
            Type == "buy" ->
                NewMarketValue = MarketValue - Quantity * MarketValue * 0.001 +
                    PendingWeigth;
            Type == "sell" ->
                NewMarketValue = MarketValue + Quantity * MarketValue * 0.001 +
                    PendingWeigth
        end,
        NewMarketValue.
```

The idea of the algorithm is to increase market value if **demand is higher**, and decrease it if **offer is higher**.

After each market value update, the oldest fillable order is searched since the new market value may have made it possible to fill limit orders.

3.3.3 WebSocket

Each client creates a WebSocket with a **single Coin Node**. When an order is filled, the Coin Node communicates the new transactions and market value using the WebSocket. Each message coming from the WebSocket contains the following fields:

- Target asset
- New market value of the target asset
- List of completed transactions. Each transaction consists of the buyer, seller, asset, quantity, old and new market value, timestamp and UUIDs of the completed orders

This message is exploited by the frontend in order to update the following data:

- Price chart of the asset
- List of assets along their market values
- List of user pending orders
- List of last transactions

3.4 Load Balancer

To make sure that the client can communicate with one of the available coin nodes, an *Nginx server* has been deployed as a **load balancer** and **reverse proxy**. The balancing policy chosen is *least_connection*, which selects the Coin Node with the least number of active connections. This system allows, on the client side, to be able to know only the IP address of the load balancer in order to carry out any communication with the Coin Nodes. The configuration used for the Nginx server is as follows.

```
map $http_upgrade $connection_upgrade {
    default upgrade;
    '' close;
}

upstream coin-node {
```

```
    least_conn;
    server 172.24.0.11:8080;
    server 172.24.0.12:8080;
    server 172.24.0.13:8080;
}

server {
    listen 80;

    location /websocket {
        proxy_pass http://coin-node/websocket;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection $connection_upgrade;
        proxy_set_header Host $host;
    }

    location /api {
        proxy_pass http://coin-node/api;
    }
}
```

3.5 Web App

The web app consists of three pages: the login and registration page, the trade page, and the user page.

Most user interactions occur on the Trade page which depends on the asset selected by the user. Messages received through WebSockets **dynamically modify** the page without requiring it to be reloaded. Considering the dynamic nature of the Trade page and the low total number of pages, *ReactJS* has been chosen for the development of the web app. This *Javascript* framework produces at build time a static single page application in *HTML*, *CSS*, and *Javascript*. The web app is served by an *Nginx server* with the following configuration.

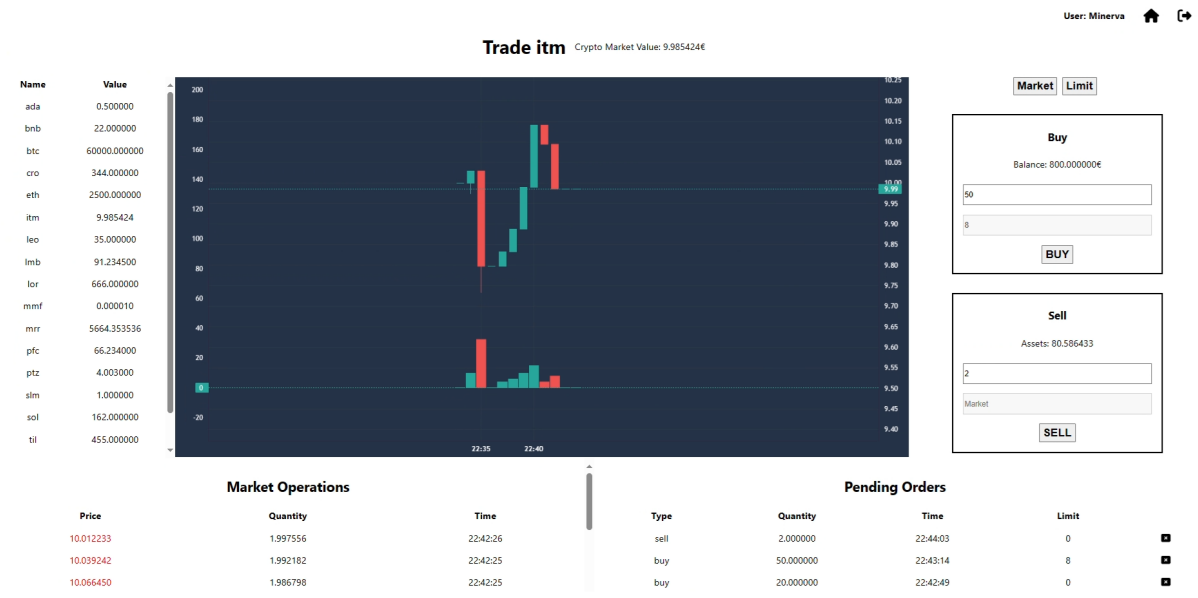
```
server {
    listen 80;

    root /usr/share/nginx/html;

    location /img {
        try_files $uri /img;
    }

    location / {
        try_files $uri /index.html;
    }
}
```

3.5.1 Login/Signup Page

In this page a user create a new account and can login in the application.

3.5.2 Trade Page

This is the most important page of the application. The chart at the center shows the evolution of the selected asset market value in real time. In particular, the upper part of the chart shows the market value trend whereas the lower part shows the relative volumes (quantity of assets traded). The two forms on the right allows the user to place a sell or buy order. On the left, instead, there is the list of assets and their market value. By clicking on one of these assets, the Trade page will change showing information about it. Finally, the two panels under the chart show the last transaction and the pending orders of the user for the selected asset.



3.5.3 User Page

In this page, the user can deposit or withdraw euros or assets and see a summary of his/her wallet.

Minerva

My Balance

500.000000€

euro 0

DEPOSIT WITHDRAW

My Assets

Coin	Quantity
leo	4567.567000
btc	0.003450
ada	25.000000
itm	120.000000
eth	12.567000