**Задание 1.**

Реализация одноранговой (peer-to-peer) сети для обмена сообщениями.

Каждый узел сети представляет собой запущенное приложение. Опционально (по желанию), можно использовать для каждого узла отдельную виртуальную машину.

Каждый узел сети знает адреса остальных узлов, может отправить запрос (текстовое сообщение) любому другому узлу (подобно клиенту) и принять запрос от другого узла (подобно серверу).

Для организации сети можно использовать сокеты (socket) непосредственно или более абстрактные объекты.

Для реализации одноранговой сети на Node.js для обмена сообщениями можно использовать модуль Socket.io. Socket.io позволяет установить соединение между клиентами и сервером в режиме реального времени, обмениваться данными и управлять событиями.

Для начала устанавливаем модуль Socket.io:

npm install socket.io

Далее, создаём сервер:

const express = require('express');

const app = express();

const http = require('http');

const server = http.createServer(app);

app.use(express.static(\_\_dirname + '/public'));

server.listen(3000, () => {

console.log('Server started on port 3000');

});

const io = require('socket.io')(server);

Установка обработчиков событий:

io.on('connection', (socket) => {

console.log('New user connected');

socket.on('disconnect', () => {

console.log('User disconnected');

});

socket.on('chat message', (msg) => {

console.log('message: ' + msg);

io.emit('chat message', msg);

});

});

Событие `connection` обрабатывает подключение нового пользователя, событие `disconnect` - отключение пользователя. Событие `chat message` принимает сообщение от клиента и отправляет его всем подключенным пользователям.

В клиенте нужно подключить Socket.io и отправлять сообщения на сервер:

<script src="/socket.io/socket.io.js"></script>

<script>

const socket = io();

$('form').submit(() => {

socket.emit('chat message', $('#m').val());

$('#m').val('');

return false;

});

socket.on('chat message', (msg) => {

$('#messages').append($('<li>').text(msg));

});

</script>

Полный пример кода можно найти в документации Socket.io: <https://socket.io/get-started/chat/>

Для реализации одноранговой сети в Node.js можно использовать библиотеку Peer.js. Она предоставляет инструменты для создания peer-to-peer соединений, обмена данными и совместной работы между браузерами и узлами Node.js.

Пример использования Peer.js для создания одноранговой сети для обмена сообщениями в Node.js:

1. Установка библиотеки Peer.js:

npm install peer

2. Импортирование библиотеки Peer.js и создание объекта Peer:

const Peer = require('peer').Peer;

const peer = new Peer({

host: 'localhost', // адрес хоста

port: 9000, // порт для подключения

});

3. Создание соединения с другим узлом:

const conn = peer.connect('peer-id');

conn.on('open', () => {

console.log('Connection established');

});

4. Отправка сообщений через соединение:

conn.send('Hello, World!');

5. Обработка входящих сообщений:

peer.on('connection', (conn) => {

conn.on('data', (data) => {

console.log('Received message:', data);

});

});

Peer.js включает в себя также дополнительные функции, позволяющие легко настроить порты, создавать и управлять комнатами (rooms) и т.д. Это делает ее удобным и мощным инструментом для создания peer-to-peer приложений на Node.js.

**Задание 2.**

Определить содержание и структуру транзакций и блоков. Разработать объекты для их хранения в программе.

Содержание и структура транзакций и блоков в зависимости от конкретной реализации может отличаться. Однако, обычно они имеют следующие элементы:

Транзакция:

- Hash транзакции (уникальный идентификатор транзакции)

- Входы: список ссылок на выходы других транзакций для использования в данной транзакции

- Выходы: список адресов получателей и суммы, которые они получают в результате транзакции

- Подписи: Доказательство того, что входы принадлежат вам

Блок:

- Hash блока (уникальный идентификатор блока)

- Номер блока в цепочке блоков

- Hash предыдущего блока в цепочке

- Транзакции: список транзакций, включенных в блок

- Награда за майнинг: сумма, которую получает майнер за создание блока

- Номер nonce: случайное число, используемое для поиска правильного хеша

Для хранения этих объектов в программе на node js можно использовать соответствующие классы. Например, для транзакции можно создать класс Transaction с полями hash, inputs, outputs и signatures, а для блока - класс Block с полями hash, number, previousHash, transactions, miningReward и nonce. Кроме того, может потребоваться использовать дополнительные классы для представления входов и выходов транзакций.

Структура блока:

- Index: уникальный идентификатор блока в цепочке

- Timestamp: время создания блока

- Transactions: массив транзакций, записанных в блоке

- Nonce: число, используемое при добыче блока

- Previous Block Hash: хэш предыдущего блока в цепочке

- Hash: хэш текущего блока

Структура транзакции:

- Sender: публичный ключ отправителя

- Recipient: публичный ключ получателя

- Amount: сумма перевода

- Timestamp: время создания транзакции

- Transaction Hash: хэш транзакции

Объекты для хранения блоков и транзакций на node js можно создать с помощью классов:

// Класс для хранения транзакции

class Transaction {

constructor(sender, recipient, amount) {

this.sender = sender;

this.recipient = recipient;

this.amount = amount;

this.timestamp = Date.now();

this.transactionHash = // хэш транзакции

}

}

// Класс для хранения блока

class Block {

constructor(index, transactions, previousBlockHash) {

this.index = index;

this.timestamp = Date.now();

this.transactions = transactions;

this.nonce = 0;

this.previousBlockHash = previousBlockHash;

this.hash = // хэш блока

}

}

Для вычисления хэшей можно использовать библиотеку crypto.

Для хранения блоков и транзакций в программе на node js можно использовать массивы:

const blockchain = [];

const pendingTransactions = [];

Массив `blockchain` будет содержать в себе всю цепочку блоков, а `pendingTransactions` - необработанные транзакции.

Реализовать возможность передачи блоков и транзакций между участниками сети. В рамках учебной задачи достаточно установить соединения между каждой парой участников и распространять новые транзакции и блоки широковещательной рассылкой.

Для передачи блоков и транзакций между участниками сети на node js необходимо использовать механизм сокетов.

Ниже приведен пример реализации передачи блоков и транзакций между двумя участниками сети на node js:

// Инициализация сокетов на клиенте и сервере

const net = require('net');

const server = net.createServer();

server.on('connection', (socket) => {

console.log('Подключился новый клиент');

// Прием блоков и транзакций через сокет

socket.on('data', (data) => {

console.log(`Получены данные: ${data}`);

// Парсинг принятых данных

const { type, payload } = JSON.parse(data);

// Обработка принятых блоков и транзакций

if (type === 'block') {

handleBlock(payload);

} else if (type === 'transaction') {

handleTransaction(payload);

}

});

});

// Отправка блоков и транзакций через сокет

const client = net.createConnection({ port: 8080 }, () => {

console.log('Подключились к серверу');

// Отправка блоков и транзакций

client.write(JSON.stringify({ type: 'block', payload: block }));

client.write(JSON.stringify({ type: 'transaction', payload: transaction }));

});

// Функция для обработки принятых блоков

function handleBlock(block) {

console.log(`Обработка блока: ${JSON.stringify(block)}`);

}

// Функция для обработки принятых транзакций

function handleTransaction(transaction) {

console.log(`Обработка транзакции: ${JSON.stringify(transaction)}`);

}

Данный пример демонстрирует работу сокетов для передачи блоков и транзакций между клиентом и сервером на node js. Функции `handleBlock` и `handleTransaction` можно использовать для обработки принятых блоков и транзакций соответственно.

Реализовать blockchain в произвольном формате (например, БД) и механизм взаимодействия цепочки блоков на диске с программой. Примеры функций: добавить блок в цепочку, узнать текущий баланс, найти транзакцию по hash-значению.

Пример кода для блокчейна на базе БД SQLite:

const sqlite3 = require('sqlite3').verbose();

// Открытие БД

let db = new sqlite3.Database('./blockchain.db', (err) => {

if (err) {

console.error(err.message);

return;

}

console.log('Connected to the blockchain database.');

});

// Создание таблицы блоков

db.run(`CREATE TABLE IF NOT EXISTS blocks (

id INTEGER PRIMARY KEY,

hash TEXT NOT NULL,

previous\_hash TEXT NOT NULL,

timestamp INTEGER NOT NULL,

data TEXT NOT NULL,

nonce INTEGER NOT NULL

)`, (err) => {

if (err) console.error(err.message);

else console.log('Created blocks table.');

});

// Добавление блока в цепочку

function addBlockToChain(block) {

db.serialize(() => {

// Поиск последнего блока

db.get("SELECT hash, nonce FROM blocks WHERE id = (SELECT MAX(id) FROM blocks)", (err, row) => {

if (err) {

console.error(err.message);

return;

}

let previousHash = "";

let nonce = 0;

if (row) {

previousHash = row.hash;

nonce = row.nonce + 1;

}

// Добавление блока в БД

db.run(`INSERT INTO blocks (hash, previous\_hash, timestamp, data, nonce) VALUES (?, ?, ?, ?, ?)`, [block.hash, previousHash, block.timestamp, block.data, nonce], (err) => {

if (err) console.error(err.message);

else console.log(`Added block ${block.hash} to the chain.`);

});

});

});

}

// Получение всех блоков в цепочке

function getBlocksFromChain() {

let blocks = [];

db.all("SELECT \* FROM blocks", (err, rows) => {

if (err) {

console.error(err.message);

return;

}

rows.forEach((row) => {

let block = {

id: row.id,

hash: row.hash,

previousHash: row.previous\_hash,

timestamp: row.timestamp,

data: row.data,

nonce: row.nonce

};

blocks.push(block);

});

});

return blocks;

}

Механизм взаимодействия цепочки блоков на диске с программой на node js может быть реализован с помощью вызова функций `addBlockToChain()` и `getBlocksFromChain()` при добавлении нового блока и при запросе списка всех блоков соответственно. Кроме того, можно реализовать WebSocket сервер для передачи новых блоков между участниками сети. При получении нового блока от другого участника, его можно добавить в цепочку, используя функцию `addBlockToChain()`.

**Задание 3.**

Реализовать пользовательский интерфейс (в простейшем случае — консоль) с возможностью создания транзакций пользователем и получения информации о балансе (и других параметров, при желании). В рамках учебной задачи достаточно создать по одному счёту для каждого участника и при запуске приложения сообщить открытые ключи (адреса) всех остальных.

Для реализации пользовательского интерфейса с возможностью создания транзакций пользователем и получения информации о балансе на Node js можно использовать фреймворк Express.js.

Пример кода для создания сервера и вывода приветственного сообщения:

const express = require("express");

const app = express();

const port = 3000;

app.get("/", (req, res) => {

res.send("Welcome to my blockchain!");

});

app.listen(port, () => {

console.log(`Server is running on port ${port}`);

});

Для создания транзакции пользователь может отправить POST-запрос на /transaction с данными о переводе:

app.post("/transaction", (req, res) => {

const sender = req.body.sender;

const recipient = req.body.recipient;

const amount = req.body.amount;

// Создание транзакции и добавление её в очередь на подтверждение

res.send("Transaction added to queue");

});

Для получения информации о балансе можно отправить GET-запрос на /balance с указанием адреса пользователя:

app.get("/balance/:address", (req, res) => {

const address = req.params.address;

const balance = calculateBalance(address);

res.json({ balance });

});

Функция calculateBalance должна вычислять баланс пользователя на основе данных о его транзакциях.

Пример кода для запуска сервера:

const express = require("express");

const bodyParser = require("body-parser");

const app = express();

const port = 3000;

app.use(bodyParser.json());

app.get("/", (req, res) => {

res.send("Welcome to my blockchain!");

});

app.post("/transaction", (req, res) => {

const sender = req.body.sender;

const recipient = req.body.recipient;

const amount = req.body.amount;

// Создание транзакции и добавление её в очередь на подтверждение

res.send("Transaction added to queue");

});

app.get("/balance/:address", (req, res) => {

const address = req.params.address;

const balance = calculateBalance(address);

res.json({ balance });

});

app.listen(port, () => {

console.log(`Server is running on port ${port}`);

});

Не забудьте подключить модуль body-parser для парсинга запросов с телом в формате JSON.

Реализовать функцию создания транзакции. По указанной для перевода сумме необходимо найти непотраченные транзакции. В качестве выходов транзакции указать получателя и сдачу на счёт отправителя.

Вот пример функции создания транзакции на node js:

const Transaction = require('./transaction.js');

function createTransaction(unspentTransactions, senderAddress, receiverAddress, amount, fee) {

// Фильтруем непотраченные транзакции для адреса отправителя

const senderUnspentTransactions = unspentTransactions.filter((tx) => {

return tx.address === senderAddress;

});

// Находим непотраченные транзакции с достаточной суммой

let total = 0;

const inputs = [];

for (let i = 0; i < senderUnspentTransactions.length; i++) {

const tx = senderUnspentTransactions[i];

inputs.push({

txId: tx.txId,

outputIndex: tx.outputIndex,

script: senderAddress.script // скрипт отправителя

});

total += tx.amount;

if (total >= amount) {

break;

}

}

// Проверяем, что нашли достаточную сумму

if (total < amount) {

throw new Error('Not enough funds');

}

const outputs = [

{

amount: amount,

script: receiverAddress.script // скрипт получателя

},

{

amount: total - amount - fee,

script: senderAddress.script // скрипт отправителя

}

];

// Создаем транзакцию и подписываем ее

const transaction = new Transaction(inputs, outputs);

transaction.sign(senderAddress.privateKey);

return transaction;

}

В этой функции мы сначала фильтруем непотраченные транзакции для адреса отправителя. Затем мы проходим по ним и находим транзакции, которые в сумме дают необходимую сумму для перевода. Если не нашли достаточную сумму, выбрасываем исключение.

Далее мы создаем выходы транзакции, указываем адреса получателя и отправителя и подписываем транзакцию скриптом отправителя.

**Задание 4.**

Организовать хранение полученных транзакций и их проверку (корректность подписи, ссылку на существующие непотраченные транзакции)

Для хранения полученных транзакций и их проверки на Node.js можно создать объект-хранилище и функции взаимодействия с ним.

Пример кода:

class Transaction {

constructor(sender, recipient, amount) {

this.sender = sender;

this.recipient = recipient;

this.amount = amount;

}

}

class TransactionStorage {

constructor() {

this.transactions = [];

}

add(transaction) {

this.transactions.push(transaction);

}

getAll() {

return this.transactions;

}

checkValidity(transaction) {

// Проверка валидности транзакции

// Возвращает true, если транзакция валидна, иначе - false

}

}

// Пример использования

const storage = new TransactionStorage();

// Добавляем транзакцию

const transaction = new Transaction('sender\_address', 'recipient\_address', 100);

storage.add(transaction);

// Получаем все транзакции

const allTransactions = storage.getAll();

// Проверяем валидность транзакции

const isValid = storage.checkValidity(transaction);

В функции `checkValidity()` можно проводить проверку подписи отправителя, проверку наличия достаточного баланса, проверку на двойную трату и т.д.

Реализовать формирование новых блоков из полученных транзакций. В рамках учебной задачи достаточно зафиксировать количество транзакций для одного блока (например, 4 транзакции на блок). В качестве алгоритма консенсуса можно использовать Proof of Work, подбирая соответствующее поле блока до тех пор, пока hash блока не будет содержать указанное число нулей в старших битах (параметр сложности подобрать экспериментально из соображений отладки ПО).

Для формирования новых блоков из полученных транзакций на node js, необходимо выполнить следующие шаги:

1. Получить список всех неподтвержденных (незаписанных в блоки) транзакций из локальной БД.

2. Создать новый блок, который будет содержать указатель на предыдущий блок, хэш новых транзакций и другую метаинформацию, например, время создания блока.

3. Рассчитать хэш нового блока с помощью алгоритма хэширования (например, SHA-256) и добавить его в блок.

4. Записать новый блок в локальную БД.

5. Отправить новый блок на сеть для подтверждения другими участниками.

Пример кода на Node.js:

const crypto = require('crypto');

const Transaction = require('./transaction.js');

const Block = require('./block.js');

const Blockchain = require('./blockchain.js');

// Получение списка неподтвержденных транзакций

const unconfirmed\_transactions = Blockchain.getUnconfirmedTransactions();

// Создание нового блока

const previous\_hash = Blockchain.getLastBlock().hash;

const new\_block = new Block(previous\_hash);

// Добавление транзакций в новый блок

for (const transaction of unconfirmed\_transactions) {

new\_block.addTransaction(transaction);

}

// Рассчет хэша нового блока и добавление его в блок

const block\_string = new\_block.toString();

const sha256 = crypto.createHash('sha256');

sha256.update(block\_string);

new\_block.hash = sha256.digest('hex');

// Запись нового блока в локальную БД

Blockchain.addBlock(new\_block);

// Отправка нового блока на сеть для подтверждения другими участниками

Blockchain.broadcastBlock(new\_block);

Принимающий чужой блок участник должен проверить корректность PoW и транзакций, а затем добавить блок в свою цепочку.

Для реализации проверки полученного блока необходимо выполнить следующие шаги:

1. Проверить, что указанное значение хэша предыдущего блока соответствует фактическому значению хэша последнего блока в текущей цепочке.

2. Проверить, что значение nonce в блоке соответствует основанным на PoW вычислениям.

3. Проверить корректность каждой транзакции в блоке в соответствии с набором правил, определенных для конкретной реализации.

4. Если проверка прошла успешно, добавить новый блок в цепочку.

Код для выполнения этих шагов может выглядеть следующим образом:

function verifyBlock(block, lastBlock) {

// Verify previous block hash

if (block.prevHash !== lastBlock.hash) {

return false;

}

// Verify proof of work

const hash = sha256(block.prevHash + JSON.stringify(block.data) + block.nonce);

if (hash.substring(0, difficulty) !== Array(difficulty + 1).join('0')) {

return false;

}

// Verify each transaction in the block

for (const transaction of block.data) {

if (!verifyTransaction(transaction)) {

return false;

}

}

return true;

}

Здесь `difficulty` - сложность PoW, а `verifyTransaction` - функция проверки корректности транзакции. Если функция `verifyBlock` вернула значение `true`, то блок может быть добавлен в текущую цепочку.

**Напиши полную реализацию обмена криптовалютой между несколькими участниками посредством блоков и транзакций на Node js**

Код реализации обмена криптовалютой между двумя участниками:

// блоки

class Block {

constructor(index, timestamp, data, previousHash = '') {

this.index = index;

this.timestamp = timestamp;

this.data = data;

this.previousHash = previousHash;

this.hash = this.calculateHash();

this.nonce = 0;

}

calculateHash() {

return SHA256(this.index + this.previousHash + this.timestamp + JSON.stringify(this.data) + this.nonce).toString();

}

mineBlock(difficulty) {

while (this.hash.substring(0, difficulty) !== Array(difficulty + 1).join('0')) {

this.nonce++;

this.hash = this.calculateHash();

}

}

}

// транзакции

class Transaction {

constructor(fromAddress, toAddress, amount) {

this.fromAddress = fromAddress;

this.toAddress = toAddress;

this.amount = amount;

}

}

// блокчейн

class Blockchain {

constructor() {

this.chain = [this.createGenesisBlock()];

this.difficulty = 2;

this.pendingTransactions = [];

this.miningReward = 100;

}

createGenesisBlock() {

return new Block(0, new Date(), 'Genesis block', '0');

}

getLatestBlock() {

return this.chain[this.chain.length - 1];

}

minePendingTransactions(miningRewardAddress) {

let block = new Block(Date.now(), this.pendingTransactions);

block.previousHash = this.getLatestBlock().hash;

block.mineBlock(this.difficulty);

this.chain.push(block);

this.pendingTransactions = [

new Transaction(null, miningRewardAddress, this.miningReward)

];

}

createTransaction(transaction) {

this.pendingTransactions.push(transaction);

}

getBalanceOfAddress(address) {

let balance = 0;

for (const block of this.chain) {

for (const trans of block.data) {

if (trans.fromAddress === address) {

balance -= trans.amount;

}

if (trans.toAddress === address) {

balance += trans.amount;

}

}

}

return balance;

}

isChainValid() {

for (let i = 1; i < this.chain.length; i++) {

const currentBlock = this.chain[i];

const previousBlock = this.chain[i - 1];

if (currentBlock.hash !== currentBlock.calculateHash()) {

return false;

}

if (currentBlock.previousHash !== previousBlock.hash) {

return false;

}

}

return true;

}

}

// пример использования

let coin = new Blockchain();

coin.createTransaction(new Transaction('address1', 'address2', 100));

coin.createTransaction(new Transaction('address2', 'address1', 50));

console.log('Starting the miner...');

coin.minePendingTransactions('miner-address');

console.log('Balance of miner is', coin.getBalanceOfAddress('miner-address'));

console.log('Starting the miner again...');

coin.minePendingTransactions('miner-address');

console.log('Balance of miner is', coin.getBalanceOfAddress('miner-address'));

Краткое описание реализации:

- создаем классы для блоков и транзакций;

- создаем класс блокчейна, который хранит цепочку блоков и имеет возможность добавлять новые транзакции и майнить блоки;

- генерируем генезис-блок, добавляем первые транзакции;

- майним блок с последними транзакциями и начисляем награду майнеру;

- проверяем баланс майнера и майним еще один блок;

- проверяем баланс майнера после двух майнинговых транзакций.