BETA VERSION, NEED FULL REVIEW AND REFACTOR

Лабораторная работа 1

18 декабря 2018 г.

“Когда кто-то говорит: «Я хочу язык программирования, который может делать все, что ему скажу», то я даю этому человеку леденец. — Alan J. Perlis”

1 Введение

Цель: изучить структуру и работу BlockChain

2 Основные понятия

- BlockChain (блокчейн) - выстроенная по определённым правилам непрерывная последовательная цепочка блоков (связный список), содержащих информацию. Чаще всего копии цепочек блоков хранятся на множестве разных компьютеров независимо друг от друга.
-
-

3 Proof-of-work

Основная концепция блокчейна довольно проста: распределенная база данных, которая поддерживает постоянно растущий список упорядоченных записей.

Однако, многое остается непонятным, если говорить о блокчейне, так же остается много проблем, которые пытаются решить с его помощью. Это относится и к популярным блокчейн проектам, таким как Биткоин (Bitcoin) и Эфириума (Ethereum). Термин "блокчейн"обычно сильно привязан к концепции типа денежных переводов, смарт-контрактов или криптовалюты. Это делает понимание блокчейна сложнее, чем есть на самом деле.

Далее будет показана реализация простого блокчейна на языке TypeScrypt. Он был выбран так как является очень похожим на язык Solidity, который будет необходимо изучить для последней лабораторной.

3.1 Структура блока

Первый логический шаг — определиться со структурой блока. Чтобы оставить все как можно проще, включим только самое необходимое (Рис. 1):

- index : Высота блока в блокчейне.

Рис. 1: Структура блоков в блокчейне.

- data: Любые данные в блоке.
- timestamp: Временная метка.
- hash: SHA256 текущего блока
- previousHash: Ссылка на хеш предыдущего блока. Это зачение полностью определяет предыдущий блок.

Код такого блока выглядит таким образом:

```
class Block {

  public index: number;
  public hash: string;
  public previousHash: string;
  public timestamp: number;
  public data: string;

constructor(index: number, hash: string, previousHash: string, timestamp: number, data: string) {
  this.index = index;
  this.previousHash = previousHash;
  this.timestamp = timestamp;
  this.data = data;
  this.hash = hash;
}

}
```

Листинг 1: Структура блока

3.2 Хеш блока

Хэш блока является одним из наиболее важных свойств блока. Хэш вычисляется по всем данным блока. Это означает, что если что-либо в блоке изменится, исходный хэш больше недействителен. Хэш блока также можно рассматривать как уникальный идентификатор блока. Например, могут появляться блоки с одним и тем же индексом, но все они имеют уникальные хэши.

Вычисляем хэш блока, используя следующий код:

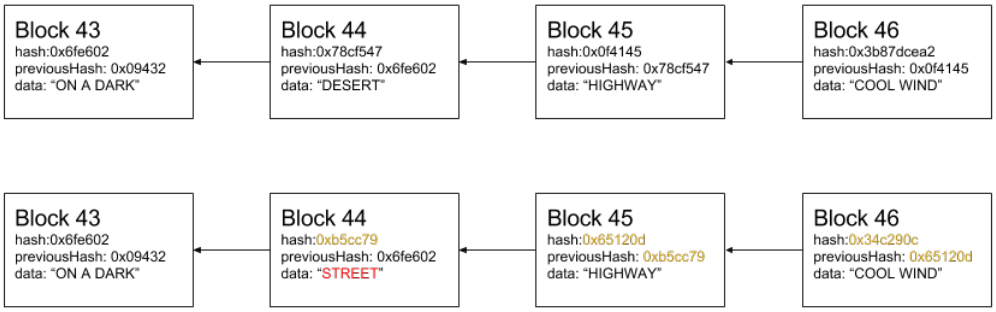

Рис. 2: Зависимость хеша от данных

```
1 const calculateHash = (index: number, previousHash: string, timestamp: number, data: string):
      string =>
2 CryptoJS.SHA256(index + previousHash + timestamp + data).toString();
```

Листинг 2: Вычисление хэша блока

Следует отметить, что блок-хэш еще не имеет ничего общего с майнингом, так как нет доказательства работы (prof-of-work). Мы используем хеши блоки, чтобы сохранить целостность блока и явно ссылаться на предыдущий блок.

Важным следствием свойств hash и previousHash является то, что блок не может быть изменен без изменения хэша каждого последовательного блока.

Это показано в приведенном ниже примере. Если данные в блоке 44 изменены с «DESERT» на «STREET», все хэши предыдущих блоков должны быть изменены (Рис. 2). Это связано с тем, что хэш блока зависит от значения предыдущего Hash (между прочим).

Это особенно важное свойство, когда вводится proof-of-work. Чем выше блок в блочной цепочке, тем сложнее его модифицировать, так как это потребует изменений для каждого предыдущего блока.

3.3 Генерация блока

Чтобы сгенерировать блок, мы должны знать хэш предыдущего блока и создать остальную часть требуемого контента (индекс, хэш, данные и временную метку). Данные блока - это то, что предоставляется конечным пользователем, но остальные параметры будут сгенерированы с использованием следующего кода:

```
1 const generateNextBlock = (blockData: string) => {
2   const previousBlock: Block = getLatestBlock();
3   const nextIndex: number = previousBlock.index + 1;
4   const nextTimestamp: number = new Date().getTime() / 1000;
5   const nextHash: string = calculateHash(nextIndex, previousBlock.hash, nextTimestamp,
        blockData);
6   const newBlock: Block = new Block(nextIndex, nextHash, previousBlock.hash, nextTimestamp,
        blockData);
7   return newBlock;
8 };
```

Листинг 3: Генерация блока

3.4 Хранение блоков

В памяти массив JavaScript используется для хранения блокчейн. Первый блок в блокчейн — это всегда так называемый «генезис-блок», имеющий следующий код:

```
const blockchain: Block[] = [genesisBlock];
```

Листинг 4: Хранение блоков

3.5 Проверка целостности блоков

В любой момент времени мы должны иметь возможность проверить, действителен ли блок или цепочка блоков с точки зрения целостности. Это верно, особенно когда мы получаем новые блоки от других узлов и должны решить, принимать их или нет.

Чтобы блок был действительным, необходимо применить следующее:

- Индекс блока должен быть на один номер больше предыдущего.
- previusHash блока соответствует hash предыдущего блока.
- hash самого блока должен быть действительным.

Это демонстрируется следующим кодом:

```
const isValidNewBlock = (newBlock: Block, previousBlock: Block) => {
  if (previousBlock.index + 1 !== newBlock.index) {
    console.log('invalid index');
    return false;
  } else if (previousBlock.hash !== newBlock.previousHash) {
    console.log('invalid previoushash');
    return false;
  } else if (calculateHashForBlock(newBlock) !== newBlock.hash) {
    console.log(typeof (newBlock.hash) + ' ' + typeof calculateHashForBlock(newBlock));
    console.log('invalid hash: ' + calculateHashForBlock(newBlock) + ' ' + newBlock.hash);
    return false;
  }
  return true;
};
```

Листинг 5: Проверка целостности блоков

Мы также должны проверить структуру блока, чтобы искаженный контент, отправленный одноранговым узлом, не разбивал наш узел.

```
const isValidBlockStructure = (block: Block): boolean => {
  return typeof block.index === 'number'
    && typeof block.hash === 'string'
    && typeof block.previousHash === 'string'
    && typeof block.timestamp === 'number'
    && typeof block.data === 'string';
};
```

Листинг 6: Проверка полей структуры блока

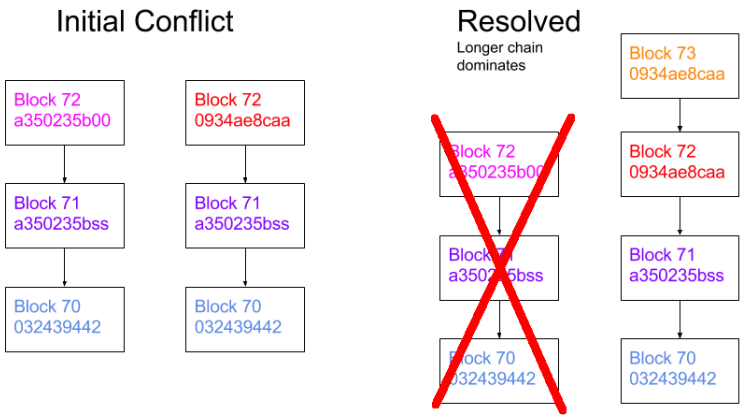

Рис. 3: Конфликт цепочек

Теперь, когда у нас есть средства для проверки одного блока, мы можем перейти к проверке полной цепочки блоков. Сначала мы проверим, что первый блок в цепочке совпадает с genesisBlock. После этого мы проверяем каждый последовательный блок с использованием ранее описанных методов. Это демонстрируется следующим кодом:

```
const isValidChain = (blockchainToValidate: Block[]): boolean => {
  const isValidGenesis = (block: Block): boolean => {
    return JSON.stringify(block) === JSON.stringify(genesisBlock);
  };

  if (!isValidGenesis(blockchainToValidate[0])) {
    return false;
  }

  for (let i = 1; i < blockchainToValidate.length; i++) {
    if (!isValidNewBlock(blockchainToValidate[i], blockchainToValidate[i - 1])) {
      return false;
    }
  }
  return true;
};
```

Листинг 7: Проверка полей структуры блока

3.6 Выбор самый длинной цепочки блоков

Всегда должен быть только один явный набор блоков в цепочке в данный момент времени. В случае конфликтов (например, два узла генерируют номер блока 72), мы выбираем цепочку с самым длинным числом блоков. В приведенном ниже примере данные (Рис. 3), введенные в блоке 72: a350235b00, не будут включены в цепочку блоков, поскольку она будет переопределена более длинной цепочкой.

Это логика реализована с использованием следующего кода:

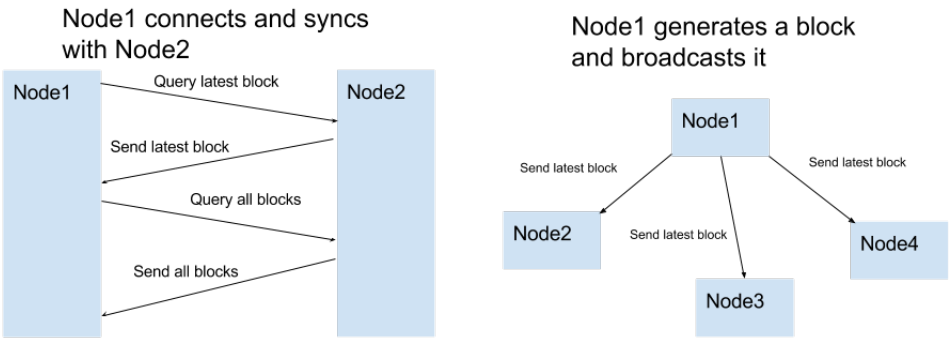

Рис. 4: Конфликт цепочек

```
1 const replaceChain = (newBlocks: Block[]) => {
2   if (isValidChain(newBlocks) && newBlocks.length > getBlockchain().length) {
3     console.log('Received blockchain is valid. Replacing current blockchain with received
          blockchain');
4     blockchain = newBlocks;
5     broadcastLatest();
6   } else {
7     console.log('Received blockchain invalid');
8   }
9 };
```

Листинг 8: Выбор самой длинной цепочки блоков

3.7 Работа с узлами сети

Существенной частью узла является совместное использование и синхронизация блочной цепи с другими узлами. Следующие правила используются для синхронизации сети(Рис. 4).

- Когда узел генерирует новый блок, он передает его в сеть.
- Когда узел соединяется с новым одноранговым узлом, он запрашивает последний блок.
- Когда узел встречает блок с индексом, большим, чем текущий известный блок, он либо добавляет блок своей текущей цепочке, либо запросы для полной блок-цепи.

Мы будем использовать websockets для одноранговой связи. Активные сокеты для каждого узла хранятся в переменной const sockets: WebSocket []. Автоматическое обнаружение одноранговых сетей не используется. Места расположения (= URL-адреса веб-узлов) пиров должны быть добавлены вручную.

3.8 Управление узлом

Пользователь должен иметь возможность управлять узлом каким-либо образом. Это делается путем настройки HTTP-сервера.

```
const initHttpServer = ( myHttpPort: number ) => {
  const app = express();
  app.use(bodyParser.json());

  app.get('/blocks', (req, res) => {
    res.send(getBlockchain());
  });
  app.post('/mineBlock', (req, res) => {
    const newBlock: Block = generateNextBlock(req.body.data);
    res.send(newBlock);
  });
  app.get('/peers', (req, res) => {
    res.send(getSockets().map(( s: any ) => s._socket.remoteAddress + ':' + s._socket.
        remotePort));
  });
  app.post('/addPeer', (req, res) => {
    connectToPeers(req.body.peer);
    res.send();
  });

  app.listen(myHttpPort, () => {
    console.log('Listening http on port: ' + myHttpPort);
  });
};
```

Листинг 9: Управление узлом сети

Как видно, пользователь может взаимодействовать с узлом следующими способами:

- Получить список всех блоков.
- Создать новый блок с заданным содержимым.
- Вывести или добавить пиров.

Наиболее простым способом управления узлом является, например, с Curl:

```
$ curl http://localhost:3001/blocks
```

Листинг 10: Отправка HTTP запроса, для получения списка блоков.

3.9 Архитектура

Следует отметить, что узел фактически предоставляет два веб-сервера: один для пользователя для управления узлом (HTTP-сервер) и один для одноранговой связи между узлами. (HTTP-сервер Websocket)

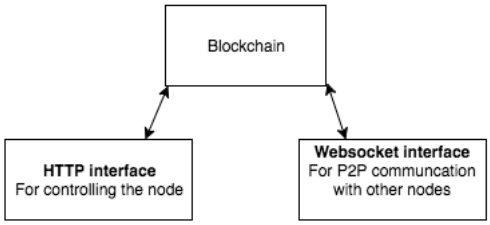

Рис. 5: Архитектура.

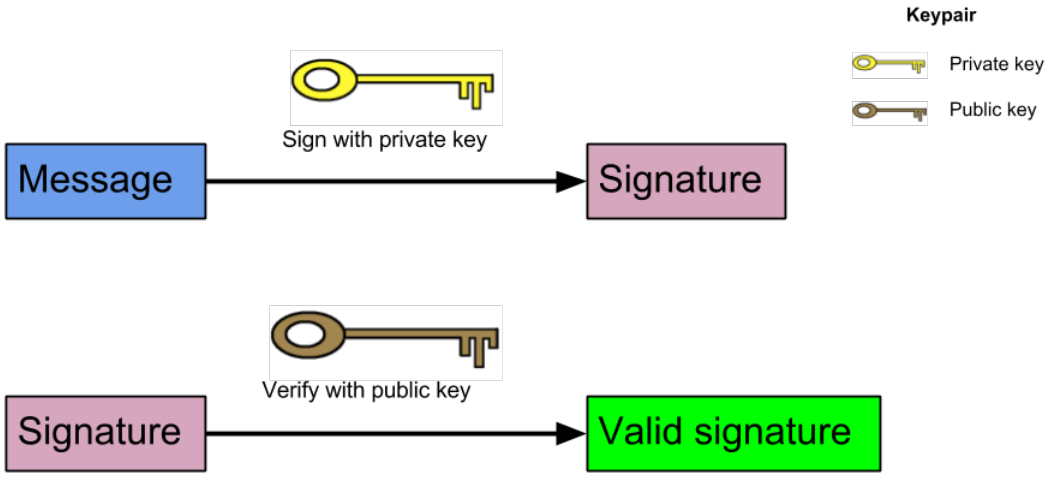

Рис. 6: Подпись с использованием открытого и закрытого ключа.

4 Транзакции

4.1 Криптография на открытых ключах и подпись

В криптографии с открытым ключом у вас есть ключевая пара: секретный ключ и открытый ключ. Открытый ключ может быть получен из секретного ключа, но секретный ключ не может быть получен из открытого ключа. Открытый ключ (как следует из названия) может быть безопасно передан всем.

Любые сообщения могут быть подписаны с использованием закрытого ключа для создания подписи. С помощью этой подписи и соответствующего открытого ключа каждый может проверить, что подпись создается секретным ключом, который соответствует открытому ключу (Рис. 6).

Мы будем использовать библиотеку, называемую elliptic для криптографии с открытым ключом, которая использует эллиптические кривые. ECDSA

В криптовалютах для разных целей используются две разные криптографические функции:

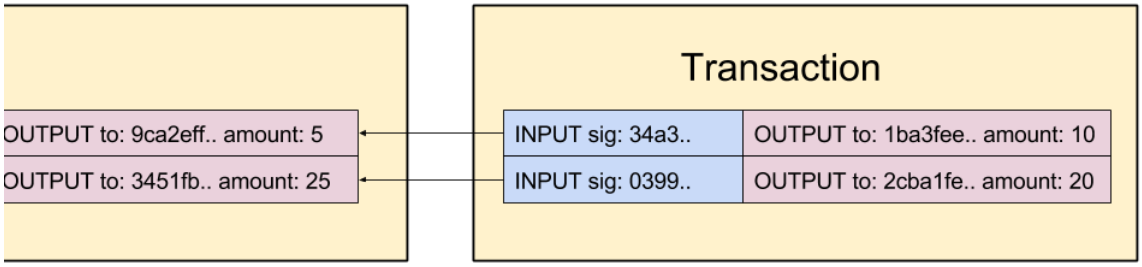

Рис. 7: Структура транзакции.

- Хеш-функция (SHA256) для интеллектуального анализа (хэш также используется для сохранения целостности блока).
- Криптография с открытым ключом (ECDSA) для транзакций.

4.2 Открытые и закрытые ключи в ECDSA

Допустимым закрытым ключом является любая случайная 32-байтовая строка, например:

```
19f128debc1b9122da0635954488b208b829879cf13b3d6cac5d1260c0fd967c
```

Допустимым открытым ключом является "04 объединенный с 64-байтовой строкой, например:

```
04bfcab8722991ae774db48f934ca79cfb7dd991229153b9f732ba5334aafcd8e7266e47076996b55
a14bf9913ee3145ce0cfc1372ada8ada74bd287450313534a
```

Открытый ключ можно получить из закрытого ключа. Открытый ключ будет использоваться как "получатель"(адрес) монет в транзакции.

4.3 Знакомство с транзакциями

Прежде чем писать код, давайте рассмотрим структуру транзакций. Транзакции состоят из двух компонентов: вводы и выводы. В выводе указывается, куда отправляются монеты, а ввод дает доказательство того, что монеты, которые действительно отправлены, существуют и в первую очередь и принадлежат "отправителю". Вводы всегда относятся к существующему выводу (Рис. 7).

4.4 Выводы транзакций

Выводы транзакции txOut состоят из адреса и количества монет. Адрес является открытым ключом ECDSA. Это означает, что пользователь, имеющий закрытый ключ ссылочного открытого ключа (адрес), сможет получить доступ к монетам.

```
class TxOut {
  public address: string;
  public amount: number;

  constructor(address: string, amount: number) {
```

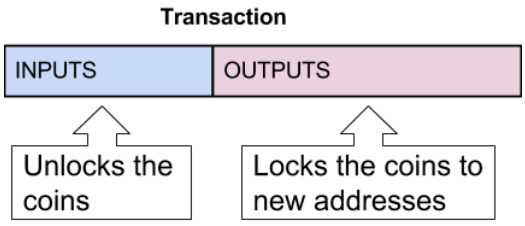

Рис. 8: Вводы и выводы.

```
6       this.address = address;
7       this.amount = amount;
8     }
9   }
```

Листинг 11: Класс вывода транзакции.

4.5 Вводы транзакций

Вводы транзакций txIn предоставляют информацию "откуда"монеты. Каждый txIn ссылается на более ранний вывод, из которого монеты «разблокированы», с подписью. Эти разблокированные монеты теперь доступны для txOut. Подпись дает доказательство того, что только пользователь, имеющий закрытый ключ упомянутого открытого ключа (адрес), мог создать транзакцию.

```
1  class TxIn {
2    public txOutId: string;
3    public txOutIndex: number;
4    public signature: string;
5  }
```

Листинг 12: Класс вводы транзакции.

Следует отметить, что txIn содержит только подпись (созданную закрытым ключом), а не сам закрытый ключ. Блок-цепочка содержит открытые ключи и подписи, а не закрытые ключи.

Можно также представить, что txIn разблокирует монеты, а txOuts заблокирует их обратно (Рис 8):

4.6 Класс транзакции

Сама структура транзакций довольно проста, так как теперь мы определили txIns и txOuts.

```
1  class Transaction {
2    public id: string;
3    public txIns: TxIn[];
4    public txOuts: TxOut[];
```

```
}
```

Листинг 13: Класс транзакции.

4.7 Идентификатор транзакции

Идентификатор транзакции рассчитывается путем принятия хеша из содержимого транзакции. Однако подписи txIds не включаются в хэш транзакции, поскольку они будут добавлены позже к транзакции.

```
const getTransactionId = (transaction: Transaction): string => {
  const txInContent: string = transaction.txIns
    .map((txIn: TxIn) => txIn.txOutId + txIn.txOutIndex)
    .reduce((a, b) => a + b, '');

  const txOutContent: string = transaction.txOuts
    .map((txOut: TxOut) => txOut.address + txOut.amount)
    .reduce((a, b) => a + b, '');

  return CryptoJS.SHA256(txInContent + txOutContent).toString();
};
```

Листинг 14: Генерирование идентификатора транзакции.

4.8 Подпись транзакции

Важно, чтобы содержимое транзакции не могло быть изменено после его подписания. Поскольку транзакции являются общедоступными, каждый может получить доступ к транзакциям, даже до того, как они будут включены в блок-цепочку.

При подписании транзакционных входов будет подписан только txId. Если какое-либо содержимое в транзакции изменено, txId должен измениться, что делает транзакцию и подпись недопустимыми.

```
const signTxIn = (transaction: Transaction, txInIndex: number,
  privateKey: string, aUnspentTxOuts: UnspentTxOut[]): string => {

  const txIn: TxIn = transaction.txIns[txInIndex];
  const dataToSign = transaction.id;
  const referencedUnspentTxOut: UnspentTxOut = findUnspentTxOut(txIn.txOutId, txIn.
      txOutIndex, aUnspentTxOuts);
  const referencedAddress = referencedUnspentTxOut.address;
  const key = ec.keyFromPrivate(privateKey, 'hex');
  const signature: string = toHexString(key.sign(dataToSign).toDER());
  return signature;
};
```

Листинг 15: Подписание входа транзакции.

Попробуем понять, что произойдет, если кто-то попытается изменить транзакцию:

1. Атакующий запускает узел и получает транзакцию с контентом: «отправьте 10 монет с адреса AAA на BBB» с txId 0x555 ..

2. Злоумышленник изменяет адрес получателя на CCC и передает его по сети в сети. Теперь содержимое транзакции «отправить 10 монет с адреса AAA в CCC»

3. Однако, когда адрес получателя изменен, txId больше недействителен. Новый действительный txId будет 0x567 ...

4. Если значение txId установлено на новое значение, подпись недействительна. Подпись соответствует только оригинальному txId 0x555 ..

5. Модифицированная транзакция не будет приниматься другими узлами, поскольку в любом случае она недействительна.

4.9 Неизрасходованные выходы транзакций

Вход транзакции должен всегда ссылаться на неизрасходованные выход транзакции (uTxO). Следовательно, когда вы владеете некоторыми монетами в блокчейне, то, что у вас есть на самом деле, это список неизрасходованных выходов транзакции, открытый ключ которых соответствует вашему закрытому ключу.

Что касается проверки транзакций, мы можем сосредоточиться только на списке выводов нерассмотренных транзакций, чтобы выяснить, действительна ли транзакция. Список неизрасходованных выходов транзакций всегда можно вывести из текущей блок-цепи. В этой реализации мы будем обновлять список неизрасходованных выходов транзакций при обработке и включении транзакций в цепочку.

Структура данных для неизрасходованного вывода транзакций выглядит следующим образом:

```
class UnspentTxOut {
  public readonly txOutId: string;
  public readonly txOutIndex: number;
  public readonly address: string;
  public readonly amount: number;

  constructor(txOutId: string, txOutIndex: number, address: string, amount: number) {
    this.txOutId = txOutId;
    this.txOutIndex = txOutIndex;
    this.address = address;
    this.amount = amount;
  }
}
```

Листинг 16: Класс неиспользованноых выходов.

Сама структура даных это просто список обьектов UnspentTxOut:

```
let unspentTxOuts: UnspentTxOut[] = [];
```

4.10 Обновление неизрасходованных транзакционных выходов

Каждый раз, когда в блокчейн добавляется новый блок, мы должны обновлять наш список неизрасходованных выходов транзакции. Это связано с тем, что новые транзакции будут тратить некоторые из существующих транзакционных выходов и внедрять новые неизрасходованные выходы.

Чтобы сделать это, мы сначала извлечем из нового блока все новые неиспользованные выходы транзакции (newUnspentTxOuts):

```
const newUnspentTxOuts: UnspentTxOut[] = newTransactions
  .map((t) => {
    return t.txOuts.map((txOut, index) => new UnspentTxOut(t.id, index, txOut.address, txOut
        .amount));
  })
  .reduce((a, b) => a.concat(b), []);
```

Мы также должны знать, какие выходы транзакции используются для создания новой транзакции (consumedTxOuts). Это будет решаться путем изучения входов новых транзакций:

```
const consumedTxOuts: UnspentTxOut[] = newTransactions
  .map((t) => t.txIns)
  .reduce((a, b) => a.concat(b), [])
  .map((txIn) => new UnspentTxOut(txIn.txOutId, txIn.txOutIndex, '', 0));
```

Наконец, мы можем генерировать новые неиспользованные транзакционные выходы, удаляя consumedTxOuts и добавляя newUnspentTxOuts к нашим существующим выводам транзакций.

```
const resultingUnspentTxOuts = aUnspentTxOuts
  .filter(((uTxO) => !findUnspentTxOut(uTxO.txOutId, uTxO.txOutIndex, consumedTxOuts)))
  .concat(newUnspentTxOuts);
```

Описанный код и функциональность содержатся в методе updateUnspentTxOuts. Следует отметить, что этот метод вызывается только после того, как транзакции в блоке (и самом блоке) были проверены.

5 Проверка транзакций

Теперь мы можем, наконец, изложить правила, которые делают транзакцию действительной:

5.1 Правильная структура транзакции

Транзакция должна соответствовать определенным классам Transaction, TxIn и TxOut

```
const isValidTransactionStructure = (transaction: Transaction) => {
  if (typeof transaction.id !== 'string') {
  console.log('transactionId missing');
  return false;
  }
  ...
  //check also the other members of class
```

```
}
```

Листинг 17: Проверка структуры транзакции.

Идентификатор транзакции должен быть правильно рассчитан.

```
if (getTransactionId(transaction) !== transaction.id) {
  console.log('invalid tx id: ' + transaction.id);
  return false;
}
```

Листинг 18: Проверка идентификатора транзакции.

Подписи в txIns должны быть действительными, а выходы, на которые ссылаются, не должны быть использованы.

```
const validateTxIn = (txIn: TxIn, transaction: Transaction, aUnspentTxOuts: UnspentTxOut[]):
    boolean => {
  const referencedUTxOut: UnspentTxOut =
    aUnspentTxOuts.find((uTxO) => uTxO.txOutId === txIn.txOutId && uTxO.txOutId ===
        txIn.txOutId);
  if (referencedUTxOut == null) {
    console.log('referenced txOut not found: ' + JSON.stringify(txIn));
    return false;
  }
  const address = referencedUTxOut.address;

  const key = ec.keyFromPublic(address, 'hex');
  return key.verify(transaction.id, txIn.signature);
};
```

Листинг 19: Проверка входов транзакции.

Суммы значений, указанных в выходах, должны быть равны суммам значений, указанных на входах. Если вы ссылаетесь на вывод, содержащий 50 монет, сумма значений на новых выходах также должна составлять 50 монет.

```
const totalTxInValues: number = transaction.txIns
  .map((txIn) => getTxInAmount(txIn, aUnspentTxOuts))
  .reduce((a, b) => (a + b), 0);

const totalTxOutValues: number = transaction.txOuts
  .map((txOut) => txOut.amount)
  .reduce((a, b) => (a + b), 0);

if (totalTxOutValues !== totalTxInValues) {
  console.log('totalTxOutValues !== totalTxInValues in tx: ' + transaction.id);
  return false;
}
```

Листинг 20: Проверка выходов транзакции.

5.2 Транзакция - база монет

Входы транзакций должны всегда ссылаться на неизрасходованные выходы транзакции, но откуда взялись исходные монеты в блок-цепочку? Чтобы решить эту проблему, вводится специальный тип транзакции: транзакция coinbase

Транзакция coinbase содержит только выход, но не содержит входных данных. Это означает, что транзакция с монетами добавляет новые монеты в обращение. Мы определяем объем выпуска монетной базы 50 монет.

```
const COINBASE_AMOUNT: number = 50;
```

Транзакция coinbase всегда является первой транзакцией в блоке, и она включается майнером блока. Награда за монетную базу выступает в качестве стимула для майнеров: если вы найдете блок, вы сможете собрать 50 монет.

Мы добавим высоту блока для ввода транзакции базу монет. Это гарантирует, что каждая транзакция coinbase имеет уникальный txId. Без этого правила, например, транзакция с монетной базой, содержащая "дать 50 монет для адреса 0xabc всегда будет иметь одинаковый txId.

Валидация транзакции с коинбазой немного отличается от проверки "нормальной"транзакции:

```
const validateCoinbaseTx = (transaction: Transaction, blockIndex: number): boolean => {
  if (getTransactionId(transaction) !== transaction.id) {
    console.log('invalid coinbase tx id: ' + transaction.id);
    return false;
  }
  if (transaction.txIns.length !== 1) {
    console.log('one txIn must be specified in the coinbase transaction');
    return;
  }
  if (transaction.txIns[0].txOutIndex !== blockIndex) {
    console.log('the txIn index in coinbase tx must be the block height');
    return false;
  }
  if (transaction.txOuts.length !== 1) {
    console.log('invalid number of txOuts in coinbase transaction');
    return false;
  }
  if (transaction.txOuts[0].amount != COINBASE_AMOUNT) {
    console.log('invalid coinbase amount in coinbase transaction');
    return false;
  }
  return true;
};
```

Листинг 21: Проверка транзакции coinbase.

Список литературы

[1]