Программирование узла блокчейн

Коваленко Г. А.

СОДЕРЖАНИЕ

1. Введение

Блокчейн - это непрерывная цепочка блоков (односвязный список), имеющая лишь две операции: чтение и добавление, исключая при этом функции редактирования и удаления за счёт элементов криптографии и компьютерных сетей. Необходимость такого способа хранения информации обусловлена в ситуациях, когда участники протокола не доверяют посредникам и друг-другу (обмен без участия арбитра, транзакция средств без участия банка).

Реализовывать блокчейн можно разными способами, даже если сфера применения заранее известна. Так например, если разрабатываемая программа на основе блокчейна является платёжной системой, то получение баланса пользователя уже можно реализовать двумя способами: детерминированным и недетерминированным*, не говоря уже о том, из чего будет состоять блок, каковы ограничения блока, какие награды за майнинг и тд. Такое состояние дел несёт более негативный характер, так как безопасность итогового продукта не будет определяться общепринятыми стандартами, которые прошли открытый и долговременный анализ. Чтобы бороться с этим фактором принято придерживаться стандартов де-факто, подобия биткоина «Bitcoin» (который выступает в качестве классической платёжной системы) и эфириума «Ethereum» (в качестве платформы контрактов).

*Детерминированность можно рассматривать как математическую функцию, результат которой зависит только от принятых аргументов и не от чего более. Её также называют концепцией «без состояний», отличной аналогией могут служить функциональные языки программирования (Haskell, LISP и тд.) [1]. Так например, чтобы получить баланс пользователя, необходимо будет перебрать всё множество блоков, которое относится к принятию / отправлению монет. Тем самым, создастся вся последовательность операций человека и полная закономерность его действий от начала и до конца.

Недетерминированность можно рассматривать как концепцию «с состояниями», и аналогией для этого могут служить императивные языки

программирования (C, Go и тд.). Если необходимо перенести это свойство на блокчейн, то можно сделать так, чтобы состояние баланса пользователя сохранялось в отдельном блоке, а не в цепочке блоков. Таким образом, чтение баланса пользователя будет происходить с конца цепочки и прекратится в тот момент, когда найдётся нужный блок.

2. Теория

2.1. Блоки

Стоит начать теоретические рассуждения по поводу блокчейна с минимальной его составляющей, а именно блока. Каждый отдельный блок можно ассоциировать с атомом физического мира, который является относительно неделимым, но состоит при этом из протонов, нейтронов, электронов, а если заглянуть глубже то и до кварков можно дойти. Такая же ситуация характерна для блока, он состоит из списка транзакций, текущего и предыдущего хеша, сложности вычисления, метки времени, подписи майнера и может содержать ещё больше информации (как например состояние баланса для недетерминированной концепции). И этот раздел предназначен в первую очередь для анализа внутренностей блока.

Стоит начать рассмотрение блока с самой минимальной его версии, где он способен оставаться и функционировать как объект в блокчейне. Так для блока достаточно всего одного поля - хеша предыдущего блока. Эта урезанная версия проста для понимания, но и даёт при этом начало теоретической базы.

Так например, представим, что у нас существует некий блок B_0, который представляет начало цепочки, по терминологии он является генезис-блоком. Суть его в том, что он не имеет предыдущего блока и является единственным прародителем всех последующих. Можно представить блок B_0 в следующем виде:

B_0 = <случайная строка>

Случайная строка в генезис-блоке заменяет хеш блока и она в теории может быть любой, даже не случайной, так как это не влияет на безопасность. Все последующие блоки будут генерироваться исходя из следующей формулы:

$B_i = hash(B_{i-1})$,

где hash - криптографическая хеш-функция* [2, с.84][2, с.595][3, с.104]

Таким образом, получается цепочка хешей, которую лишь с оговорками можно назвать блоком. Чтобы придать блоку «смысла», нужно поместить в блок данные (D) для хранения, но таким образом, чтобы эти данные зафиксировались в результате хеша. Теперь блок усложнился и вместо одного значения (хеша), он стал хранить два (данные и хеш), тем самым став массивом:

B_0 = [<NULL>, <случайная строка>]

B_i = [D, hash(D ‖ B_{i-1}{2})]

В данном примере были введены запись «‖», что означает конкатенацию строк и запись «B_{i-1}{2}», что означает взять из массива B_{i-1} второй элемент (индексация для массива будет начинаться с единицы).

Эта запись уже более похожа на блокчейн, потому что хранит хоть какие-то значения, предполагаемо полезные пользователю. Но может возникнуть логичный вопрос: для чего это всё было сделано? И ответ - для безопасности (хоть на данном этапе ещё есть масса проблем). Суть в том, что теперь невозможно изменить значение B_i{1} (или B_{i-1}{2}) не затрагивая B_i{2}, иначе изменение будет обнаружено, потому что значение блока не будет соответствовать его хешу.

Ситуация для злоумышленника будет плачевной в тот момент, когда ему необходимо будет изменить значение блока в середине или начале цепочки, так как ему придётся изменять все последующие хеши, которые идут после изменяемого блока. Но проблемы в этой схеме всё же остаются.

Что если в блокчейне хранятся переводы денежных средств? Будет очень обидно, если злоумышленнику удастся изменить ваше значение, хранимое в блоке, на значение куда б*о*льших размеров, либо сгенерирует новый блок, в котором укажет вас как отправителя.

Чтобы решить данный недостаток, необходимо ознакомиться с таким термином как цифровая подпись** [2, с.91][2, с.663][3, с.46].

Цифровая подпись помогает идентифицировать создателя блока, тем самым исключая моменты, когда кто-либо может выдать себя за другого

человека. Сама концепция цифровой подписи несёт ещё два термина - адрес (публичный ключ) и кошелёк (приватный ключ).

Подпись должна применяться на хеш блока, а не на его значение по двум основаниям. Во-первых, хеш всегда имеет фиксированную длину, соответственно и скорость подписания не будет варьироваться. Во-вторых, подпись будет затрагивать также информацию о хеше предыдущего блока, что полностью свяжет подпись с предыдущими блоками.

Блок усложнился до трёх элементов (значение, хеш, подпись):

B_0 = [<NULL>, <случайная строка>, <NULL>]

B_i = [D, hash(D || B_{i-1}{2}), sign(B_i{2})]

Теперь, если злоумышленник захочет изменить значение B_i{1}, ему необходимо будет изменить и значение B_i{2}, но если он изменит значение B_i{2} он не сможет никак изменить значение B_i{3} так, чтобы этот блок продолжал идентифицироваться как до изменения. Или иными словами, посредник будет способен изменить блок, но только при условии того, что он поставит свою подпись на блок и точно также изменит все последующие блоки. Таким образом, если злоумышленник будет проводить подобные махинации в платёжной системе, то банкротом он станет быстро.

Но что если злоумышленник до такой степени отчаялся, что готов потратить все свои деньги лишь бы стало плохо другим? Даже если ваш блок будет находиться в начале цепочки, а цепочка в миллион блоков, современному компьютеру будет под силу изменить значение в цепочке и далее сгенерировать все последующие хеши с подписями злоумышленника, тем самым стерев все настоящие блоки.

В данном случае, на помощь приходит proof-of-work*** [4, с.216]. Это ещё б*о*льшее усложнение блока, которое добавляет такой термин как сложность вычисления. Сложность эта основана на математически трудной задаче, подобия нахождения нужного хеша. Обычный компьютер может тратить на такие задачи по несколько часов, дней, а то и месяцев и лет, не говоря уже о возможности человека найти собственноручно решение. Чтобы регулировать

сложность, была придумана задача вычисления не всего хеша, а лишь его части. Допустим, перебирается число и оно хешируется, нужно найти такое хешируемое значение, чтобы оно начиналось на четыре нулевых байта. Если такой хеш находится, значит задача решена. Но здесь определённо существует минус, если хоть один человек найдёт хешируемое число, которое заведомо будет начинаться с большого количества нулей, то придётся изменить способ вычисления трудной задачи, например перейти на единицы, потом на двойки, тройки, четвёрки и тд. Это явно неудобно, соответственно был придуман усовершенствованный способ этой задачи, что если хешировать не просто инкрементируемое число, а хешировать конкатенацию инкрементируемого числа с текущим хешем блока? Будет получаться всегда случайный результат и нельзя будет использовать полученное инкрементируемое число повторно на другом блоке. Поиск нужного хеша ещё называется майнингом.

Теперь блок будет выглядеть следующим образом:

B_0 = [<NULL>, <случайная строка>, <NULL>, <NULL>]

B_i = [D, hash(D ‖ B_{i-1}{2}), sign(B_i{2}), pow(B_i{2})], где pow возвращает число x = B_i{4}, при котором hash(x || B_i{2}) = «0000...».

Следует отметить, что функция pow сосредоточена именно на хеш. То-есть, блок не содержит никакой информации о том, кто майнил. Представим такую ситуацию, что вы сами не майните блок, а перекладываете эту задачу на группу других людей, мощность компьютеров которых в разы превышает вашу ЭВМ и за это вы платите некий процент от перечисленных средств. Далее один человек из этой группы наконец-то находит нужное число и заносит его в блок. Но в этой группе есть злоумышленник, он копирует число x и пересылает его вам. Теперь вам поступило два ответа о найденном хеше, кому доверять? Можно сказать, кто первый принёс, тот и получит награду, но можно поступить другим, более правильным способом. Пусть сам майнер будет создавать блок, а вы ему будете отправлять лишь транзакцию (а именно три первых поля блока), далее майнер будет вносить свой хеш и подпись в блок, и на свой хеш будет производить операцию pow.

Блок ещё больше усложняется:

B_0 = [<NULL>, <NULL>, <случайная строка>, <NULL>, <NULL>, <NULL>]

B_i = [[D, hash(D), sign(B_i{1}{2})], M, hash(M || B_i{1}{2}|| B_{i-1}{3}), $sign_M$(B_i{3}), pow(B_i{3})]

Когда майнер найдёт нужное число x = pow(B_i{3}), никто более не сможет скопировать это число и подставить под свой результат, так как полученное значение не будет подходить под хеш другого блока, где будет указан другой майнер. Но и при этом, необходимая транзакция попадает в блокчейн, со значением, хешем значения и подписью хеша. То-есть, всё как нужно.

Но и на данном этапе существует лазейка. Можно заменить без последствий sign(B_i{1}{2}) и никто этого не заметит. Вопрос конечно может стоят иначе, ради каких целей существующую подпись стороннего человека захочется изменять на свою подпись, при этом подстраиваясь под его хеш? Но увы всякое бывает и потому это также следует считать уязвимостью. В исправлении данной уязвимости может помочь раскрытие внутренностей волшебного значения D.

И опять блок начинает усложняться:

B_0 = [<NULL>, <NULL>, <случайная строка>, <NULL>, <NULL>, <NULL>]

B_i = [[[S, R, V], hash(B_i{1}{1}), $sign_S$(B_i{1}{2})], M, hash(M || B_i{1}{2} || B_{i-1}{3}), $sign_M$(B_i{3}), pow(B_i{3})]

В данном случае, S - sender (отправитель), R - receiver (получатель), V - value (само значение или количество денег). Также изменился вид sign(B_i{1}{2}) на $sign_S$(B_i{1}{2}), указывая на то, чья это подпись. Теперь подпись нельзя заменить, не затрагивая хеша. Изменить создателя подписи тоже нельзя, потому что необходимо будет менять хеш-значение, что неминуемо ведёт опять к замене подписи. Подпись майнера также

проблематично изменить, потому что его имя указано в хеше. Доказательство работы украсть также не получится, потому что оно привязано к хешу. Кажется будто вот она, идеальная защита! Но увы нет, ещё остались уязвимости.

Помните, что в блокчейне нельзя доверять никому? Собственно это справедливо и для майнеров. Но каким образом майнер способен навредить? Ведь была сделана подпись на хеш, а хеш указывает на отправителя, получателя и значение, то-есть ничего из этого нельзя изменить, а если и изменишь то всё полностью, что аналогично простому игнорированию добавления транзакции в блок. Это всё верно, но есть одна деталь, которая очень надоедливая - транзакция ни к чему не прикреплена. Соответственно, майнер или кто-либо другой сможет скопировать полностью транзакцию и продублировать её заново, при этом она также будет валидна, потому что все хеши и подписи правильные (ведь они и вправду настоящие). Чтобы избавиться от этой уязвимости, можно просто указать в транзакции хеш предыдущего блока.

Блок ещё немного усложнился:

B_0 = [<NULL>, <NULL>, <случайная строка>, <NULL>, <NULL>, <NULL>]

B_i = [[[S, R, V], hash(B_i{1}{1} ‖ B_{i-1}{3}), $sign_S$(B_i{1}{2})], M, hash(M ‖ B_i{1}{2} ‖ B_{i-1}{3}), $sign_M$(B_i{3}), pow(B_i{3})]

Теперь даже если майнер захочет продублировать транзакцию, она будет находиться под новым блоком, а хеш предыдущего блока уже не будет совпадать с хешем, который находится в транзакции. А если майнер попытается оставить изменённые значения, даже несмотря на несоответствие, то другие майнеры будут просто отвергать его блок как невалидный.

И вот теперь у нас блоки выстраиваются в цепочку, все они защищены от подмены, изменения, дублирования и при чём от всех пользователей. Но что если, существует несколько майнеров и у каждого из майнеров разная цепочка блоков? Как они будут согласовывать общий блокчейн? И возможно ли это? Ответом будет конечно же да, иначе не существовали бы биткоины, да

эфириумы. Вопрос скорее лежит в сложности такого изменения. Основной способ решения таких конфликтов сводится к общей сложности самой цепочки, которая была составлена из множества операций pow. Если одна цепочка превышает другую по сложности, то она и становится лидирующей. Если в цепочках сложность является статичной величиной, тогда цепочки измеряются в количестве блоков, и так цепочка, размер которой больше, будет являться лидирующей и узел с меньшей цепочкой должен уступить и принять б*о*льшую. Такие противоречащие ситуации называются soft fork, когда блокчейн разделяется на несколько ветвей, но при этом ещё существует возможность выбрать только одну «правильную» цепь. Если злоумышленник захочет подменить весь блокчейн, то ему придётся сгенерировать самостоятельно больше блоков, чем имеется в атакуемой цепочке, соответственно, если цепочка уже является огромной, то потенциал такой атаки стремится к нулю.****

И помимо soft fork’ов существуют ещё так называемые hard fork’и. Hard fork’и, в отличие от soft fork’ов, сами по себе не появляются. Они используются тогда, когда необходимым является изменение или улучшение приложения, или же создание нового блокчейна на основе уже имеющегося. Это приводит к тому, что новая ветка никак не будет «срастаться» со старой, тем самым становясь несовместимой.

Когда идёт рассуждение о транзакциях в основе которых лежат деньги, автоматически и логически возникает вопрос, а от куда возникают корни этих денег в цифровом мире блокчейна? Надеюсь вы ещё не забыли, что существует такой генезис-блок. Ну так вот, кто создаёт генезис-блок, тот и является человеком с деньгами, он способен ими манипулировать или иными словами совершать первоначальные транзакции. Но, помимо всего прочего, в блокчейне также существует хранилище, которое является неким блокчейн-пользователем, но не имеющим приватного ключа. Это можно рассматривать как исключение из правил, которое формирует новое правило только для него в контексте майнеров. Суть хранилища сводится к тому, чтобы оно выплачивает из своей «казны» деньги за выполненную работу майнерам. При этом,

хранилище может быть как самовосстанавливающимся, так и невосстанавливающимся.

Если мы говорим о невосстанавливающихся хранилищах:

1. Майнер замайнил блок
2. Хранилище платит деньги майнеру
3. Майнеру перечисляется процент с транзакций

Если мы говорим о самовосстанавливающихся хранилищах:

1. Майнер замайнил блок
2. В хранилище перечисляется процент с транзакций
3. Хранилище платит деньги майнеру

Таким образом, зная всё это, необходимо изменить генезис-блок:

B_0 = [[[S, M, V], <NULL>, <NULL>], <NULL>, <случайная строка>, <NULL>, <NULL>, <NULL>],

где S - хранилище, M - майнер генезис-блока, V - награда за генезис-блок. Как только генезис-блок сгенерировался, начинает действовать хранилище, у которого определённый и ограниченный запас ресурсов.

Кажется, словно уже обо всём сказано, но существует ещё ряд тонкостей в блокчейне.

Во-первых, если разным майнерам посылается одновременно транзакция, то у майнеров возникает конкуренция или гонка по майнингу. Шансов выиграть больше у того, кто обладает б*о*льшими мощностями, но вопрос в другом, как координируются майнеры? Ведь в теории майнеры могут майнить одни и те же значения, тем самым расходуя понапрасну время и энергию. Для такого случая, создаются пул-сервера, к которым подключается группа майнеров заинтересованных в решении одной задачи и распределении ресурсов между собой. Пул-сервера координируют действие каждого отдельного майнера, таким образом, чтобы они не пересекались в переборе одинаковых значений.

Во-вторых, транзакций в блоке может быть несколько (и так чаще всего бывает). В таком случае возникает ещё одна уязвимость, а именно

дублирование транзакции в одном блоке. Чтобы избежать этого, нужно вставлять в каждую транзакцию случайное число (r) и хешировать его вместе со всей другой информацией.

В-третьих, можно применить временные метки (Time), которые будут свидетельствовать о времени создания блока, для его последующего отслеживания в истории и определения текущей мощности блокчейна со стороны майнеров*****. Всё это возможно лишь при условии грамотной синхронизации с сервером, представляющим время.

В итоге, последняя версия блока будет выглядеть следующим образом:

B_0 = [[[[<NULL>, S, M, V], <NULL>, <NULL>]], <NULL>, <случайная строка>, <NULL>, <NULL>, <NULL>]

B_i = [[[[r, S, R, V], hash(B_i{1}{1}{1} ‖ B_{i-1}{4}), $sign_S$(B_i{1}{1}{2})], [...], ...], M, Time, hash(M ‖ Time ‖ (B_i{1}{1}{2} ‖ B_i{1}{...}{2}) ‖ B_{i-1}{4}), $sign_M$(B_i{4}), pow(B_i{4})]

*Криптографическая хеш-функция - это односторонняя функция, которая принимает строку произвольной длины и возвращает строку фиксированной длины. Свойство односторонности говорит, что легко получить y от x [y = f(x)], но сложно получить x от y [x = f^{-1}(y)]. В качестве криптографических хеш-функций рекомендуется использовать хеш-функции семейства SHA-2 (sha224, sha256, sha384, sha512) или SHA-3 (keccak). В [3, с.111] указано, что хеш-фукнции «в сыром виде» имеют недостатки, подобия удлинения сообщения и коллизий. От второго избавиться проблематично, по причине существования атаки дней рождения, тем самым принято выбирать хеш-функции б*о*льшего порядка. От первого же недостатка избавиться можно, как пример, применяя алгоритм HMAC на функцию. Но в блокчейне такая атака будет бесполезной, так как она требует редактирования уже имеющихся данных и хеша данных, что влечёт за собой необходимость редактирования подписи, по причине дальнейшего несоответствия между хешем и подписью.

**Цифровая подпись - элемент асимметричной криптографии. В таких алгоритмах шифрования как RSA является результатом функции

расшифрования, а проверка подписи осуществляется при помощи функции шифрования. Замысел в том, что функция расшифрования известна лишь создателю закрытого ключа, соответственно только он способен при помощи этого свойства подтверждать свои действия.

***PoW (Proof-of-work) - доказательство работы, осуществляемое при помощи решения сложной математической задачи (например, нахождение нужного хеша). Помимо PoW существуют и другие способы доказательства, как например PoS (Proof-of-stake), PoA (Proof-of-authority), PoB (Proof-of-burn) и тд.

****Стоит сказать, что если майнеры соберутся в одну коалицию и их суммарная мощность будет больше 50% мощности всех майнеров, то возможна атака, при которой будет возникать достаточно продолжительный soft fork. То-есть, накапливание блоков в разных цепочках будет происходить одновременно, тем самым разделяя общий блокчейн на два лагеря. Из этого явления могут выступать спекулятивные возможности отмены транзакций за счёт выбора майнерами нужной цепочки.

*****Вычислить текущую мощность майнеров можно при помощи меток времени. Если раньше на майнинг уходило в среднем 10 минут, а сейчас 5, то можно утверждать, что мощность майнинга увеличилась вдвое. При этом, тенденция к понижению необходимого времени сказывается неблагоприятным образом, так как будут возникать более частые случаи появления soft fork’ов. Из-за этого в блокчейнах предустанавливают функцию, которая регулирует текущую сложность блокчейна, подстраиваясь под константное время (например 10 минут).

2.2. Сеть

Чем больше вы изучаете сеть блокчейна, тем меньше она вам кажется полностью децентрализованной. В этой сети существуют сервера указывающие точное время, пул-сервера отвечающие за координацию действий узлов, а помимо всего прочего и сами узлы представляют из себя распределённый сервер, так как пользователи, которые осуществляют транзакции, являются клиентами.

Сутью данного раздела является выявление особенностей блокчейна со стороны сети и построение связей между её участниками. Так для начала стоит сказать, что существует два основных вида сети - многоранговые и одноранговые. Многоранговые (клиент-серверные) сети предполагают существование двух объектов - клиента и сервера, где клиент способен производить запрос, а сервер выдавать ответ. Одноранговые же (peer-to-peer) сети предполагают равноправие участников в действиях, или иными словами, пользователь в данной сети является одновременно и клиентом, и сервером (если исходить из терминологии многоранговых сетей). Таких пользователей именуют узлами.

Но помимо двух вышеперечисленных типов сетей существует некий гибрид, который так и называется - гибридная сеть. Её основная суть заключена в том, что существуют, как и в многоранговой сети, два основных объекта - клиент и сервер. Но особенность такой сети заключается в том, что в качестве сервера выступает одноранговая сеть, или иными словами сеть узлов. И это определение гибридной сети всё же может вызывать вопросы. Например, что если, какой-нибудь человек поднимет сайт, хранение данных которого будет распределено между его серверами, тогда такая сеть будет называться гибридной? И можно ли будет назвать сеть гибридной, если существует только один узел?

Не ответив на эти вопросы, вряд-ли можно продолжать рассуждение о данном теме. Соответственно, необходимо найти в многоранговых и гибридных сетях принципиальное различие, и оно существует. Узлы в гибридных сетях, в

отличие от серверов в многоранговых, не доверяют друг-другу, но и при этом выполняют общую работу. Из этого следует, что узлы не подчиняются какому-то конкретному человеку, либо узкому кругу лиц и тем самым несут децентрализованный характер. Но и из этого следует то правило, что если существует только один узел или узлы находятся в руках одной группы лиц, тогда такая сеть теряет свойства гибридной и переходит в фазу многоранговой.

В итоге, существуют следующие связи среди участников блокчейна:

1. Клиент -> [многоранговая] -> Узел
2. Узел -> [одноранговая] -> Узел
3. Узел -> [многоранговая] -> Пул-сервер
4. Узел -> [многоранговая] -> Сервер времени

1. Клиенты посылают узлам запросы на счёт получения баланса, блоков или занесения транзакции в блок.
2. Узел связывается с другими узлами для хранения общей цепочки. Узел может посылать другому узлу запросы о добавлении нового блока в блокчейн.
3. Узел запрашивает у пул-сервера необходимый диапазон майнинга.
4. Узел запрашивает у сервера времени текущее состояние времени.

Хоть количество многоранговых связей и преобладает, тем не менее, важность действий приходится на одноранговую связь. Основной вопрос здесь скорее лежит в том, ухудшается ли отказоустойчивость блокчейн сети, в моменты появления многоранговой связи?

Возьмём для начала пример пул-сервера и представим злоумышленника пытающегося его отключить от сети методом DDoS атак. Сервер отключается, к нему доступ прекращён для некоторых узлов. В итоге, эта группа узлов может пойти по двум сценариям, либо подключиться к другому пул-серверу, либо генерировать блоки по отдельности, исходя из случайного числа. В данном

случае, сеть продолжает функционировать при любом обстоятельстве, хоть и с возможными оговорками по поводу падения производительности.

Теперь же возьмём сервер времени. Отличие данного сервера от пул-сервера заключается в том, что его можно использовать лишь единожды перед запуском узла. Иными словами, запросить у сервера точное время, установить это время на локальной машине и запустить узел. Таким образом, атаки на сервер будут приносить вред лишь появлению новых узлов, в то время как уже функционирующие узлы будут продолжать работу. При этом стоит учитывать, что серверов времени всегда несколько и произвести успешную DDoS атаку представляется сложной задачей. И даже если все серверы времени были отключены от сети, в это время продолжает функционировать сам блокчейн, в узлах которых хранится текущее время. Соответственно, формирующийся узел может запросить время не напрямую, у серверов времени, а косвенно, через уже работающий узел в блокчейне.

Если подытожить всё вышесказанное, то отказоустойчивость блокчейна будет зависеть полностью от одноранговой архитектуры, при этом отключение от серверов может привести лишь к падению производительности некоторой группы майнеров, но никак не к понижению уровня отказоустойчивости.

Хорошо, если блокчейн сеть является по своей сути децентрализованной, тогда возникает логичный вопрос: каким образом клиенты (и сами узлы) будут подключаться к другим узлам? От куда они возьмут первоначальный список необходимых адресов? Это на самом деле очень частый вопрос в контексте децентрализованных сетей, так как каждая такая сеть решает подобные проблемы разными способами. Чаще всего, решением является использование сторонних каналов связи. Допустим, создаётся некий сервер, на котором будет располагаться список действующих узлов. Выведение из строя такого сервера не разрушит саму сеть, но перекроет доступ к информации о ней и её увеличению (что является проблемой лишь при начальном формировании блокчейна). Расширение списка будет происходить самими же узлами (то-есть они будут вносить свой адрес на сервер), так как заинтересованы в прибыли от майнинга. И чем чаще, на различных серверах адрес майнера будет попадаться,

тем чаще к нему будут обращаться клиенты, за помощью в майнинге (и сами майнеры, для добавления новых блоков).

Если в блокчейне учитывать добавление адресов, то количество связей в сети увеличится на две строки:

5. Клиент -> [многоранговая] -> Сервер адресов
6. Узел -> [многоранговая] -> Сервер адресов

Также стоит учитывать, что сам майнер способен выдавать список всех других майнеров, к которым он подключен, тем самым в большинстве случаев даже не нужно посещать сервера. Но может появиться один вопрос: из-за конкуренции майнеров в нахождении нужного хеша, не будет ли майнер пытаться выдавать урезанный список адресов, чтобы увеличить свои шансы на майнинг? Может и будет, но суть здесь в другом, если он не успеет замайнить блок и блокчейн обновится за счёт другого майнера, то блок, который майнил «майнер-мошенник» окажется невалидным. В итоге, ему придётся либо создать свою цепь блоков, игнорируя при этом весь другой блокчейн (что достаточно рискованно, так как это будет аналогично hard fork’у, с последующим привлечением майнеров на свой блокчейн), либо согласится с другими майнерами и принять тот факт, что транзакции от клиентов, которые посылались ему, оказались просроченными. Из этого случая клиенты поймут, что если транзакции не попали в новый блок, то скорее всего майнер выдал не весь список действующих узлов. Таким образом, клиентам лучше подключаться сразу к нескольким узлам и получать от них адреса других майнеров, отсеивая повторяющиеся.

И есть ещё третий способ, благодаря которому можно получить список узлов. Авторы программы блокчейн-клиента (либо блокчейн-узла) могут по-умолчанию внести в программу (либо в конфигурационный файл) список доверенных узлов, которые показали себя как действующие на протяжении долгого времени. И по мере обновления программы будет также обновляться этот список.

Со стороны сохранения мощности и доступности блокчейн-сети следует применять сразу все три способа нахождения адресов. Так например, если популярность данного блокчейна увеличилась на несколько порядков, то в теории первый способ нахождения адресов (при помощи сервера) может не использоваться, но тогда будет возникать уязвимость, когда доверенные узлы (третьего способа нахождения майнеров) скооперируются и не будут выдавать адреса всех других узлов, тем самым понижая мощность блокчейн-сети. Это скажется на более успешном пополнении баланса «майнеров-мошенников», при этом другие майнеры даже не получат шанса формировать новый блок, исходя из транзакций. При этом, если убрать третий способ, то будет всегда возникать необходимость обращения к серверу на инициирующем этапе запуска приложения. Если убрать первый и второй способы, то шансы занесения транзакции в новый блок будут пониженными (при условии того, что существует несколько клиентов с разным списком адресов). Если убрать второй способ, то возможность получать список новых адресов будет возложена только на сервер. Ну а если убрать первый и третий способы, то такая блокчейн-сеть будет терять лёгкость использования, из-за необходимости ручной настройки соединения.

3. Практика

Теория могла показаться сложной, но к сожалению, практика ещё сложнее. Необходимо знать программирование, чтобы воссоздать то, что было лишь идеей. В качестве языка программирования был выбран язык Go [5]. У него есть ряд преимуществ над другими языками высокого уровня, как например большая стандартная библиотека (в которую включаются криптографические алгоритмы и функции работы с сетью) и лёгкость / примитивность синтаксиса (что позволяет понимать его код, зная при этом лишь язык Си). Скачать компилятор данного языка можно с официального сайта: https://golang.org

К сожалению в Go нет встроенной СУБД (в качестве стандартной библиотеки), есть лишь интерфейсы. Таким образом, необходимо установить стороннюю открытую библиотеку, которая будет работать с СУБД SQLite3. Это можно сделать следующим образом, при помощи терминала:

go getgithub.com/mattn/go-sqlite3

Данная СУБД является встраиваемой и не будет противоречить принципу децентрализации, за счёт того, что каждый узел будет хранит копию блокчейна у себя в локальной БД.

Установив всё вышеперечисленное можно приступать к написанию кода.

Сам же код будет воспроизводиться в одном лишь файле для каждой отдельной темы, так например, библиотека для работы с сетью - один файл, библиотека для работ с блокчейном - один файл, клиентское приложение - один файл и приложение узла также один файл. Такой способ записи несёт как негативные, так и положительные моменты. Так например, это негативно сказывается на анализ всего сделанного (в готовом виде), но при этом легче воспринимается, когда реализовываешь это всё по порядку.

Код всех программ будет преподноситься в виде схемы «причина-следствие», то-есть начинаться с абстрактных функций, которые впоследствии будут нести действия, исходя из логики. Так например, будет наперёд реализовываться интерфейсная функция, а только после неё те функции,

которые находятся внутри неё. При этом также стоит сказать, что и сами подключения пакетов будут расписываться лишь в конце глав, исходя из реализованных ранее функций.

3.1. Библиотека для работы с сетью

Для создания библиотеки работающей с сетью следует использовать стандартный пакет языка Go под названием «net». Данный пакет позволит работать с TCP/IP соединением и на основе его будут созданы интерфейсные функции, а также базовый протокол передачи данных.

Передача данных будет осуществляться по принципу запрос-ответ, при этом, перед запросом будет происходить соединение, а после получения ответа - рассоединение (аналогично HTTP протоколу).

При написании любой программы на языке Go, необходимо указывать к какому пакету принадлежит данный исходный код. Так библиотеку, работающую с сетью, назовём как «network», а исходный файл под названием «network.go» расположим в директории «network/».

```
package network
```

Приступая к написанию кода надо разбраться как и какие данные будут передаваться в разрабатываемой сети. Во-первых, необходима опция (Option), например GET_BLNCE будет говорить о получении баланса, ADD_TRNSX о занесении транзакции в блок и тд. Во-вторых, нужен параметр заголовка (или сами данные), допустим при заголовке ADD_TRNSX, в качестве параметра должна будет указываться сама транзакция, которая вносится в блок. Все эти данные будут определены в структуре Package (1.1).

(1.1) Структура Package.

```
type Package struct {
	Option int
	Data string
}
```

Имея данную структуру уже можно реализовать функцию отправки данных Send (1.2), которая будет принимать адрес получателя и объект структуры Package и возвращать при этом экземпляр структуры Package (ответ от сервера).

(1.2) Функция Send.

```
func Send(address string, pack *Package) *Package {
	conn, err := net.Dial("tcp", address)
	if err != nil {
		return nil
	}
	conn.Write([]byte(SerializePackage(pack) + ENDBYTES))
	var res = new(Package)
	ch := make(chan bool)
	go func() {
		res = readPackage(conn)
		ch <- true
	}()
	 select {
		case <-ch:
		case <-time.After(WAITTIME * time.Second):
	 }
	return res
}
```

Функция создаёт TCP соединение с указанием IPv4:Port адреса при помощи функции Dial (из стандартного пакета «net»). Далее проверяется

наличие ошибки при соединении и если такая ошибка обнаруживается, тогда возвращается нулевой адрес. Далее на экземпляр pack применяется функция SerializePackage (1.3), которая переводит объект в строку. Полученная строка конкатенируется с константой ENDBYTES, свидетельствующей об окончании переданных данных (1.4). Далее при помощи метода Write полученная строка отправляется серверу, после чего функция начинает ожидать ответа от сервера, при помощи функции readPackage (1.5) на протяжении WAITTIME (1.6) секунд.

Замечание: функции с заглавной буквы - интерфейсные (extern), с прописной - внутренние (static). Это же правило справедливо для констант и переменных расположенных вне функций (в глобальном пространстве).

(1.3) Функция SerializePackage.

```
func SerializePackage(pack *Package) string {
	jsonData, err := json.MarshalIndent(*pack, "", "\t")
	if err != nil {
		return ""
	}
	return string(jsonData)
}
```

Преобразует объект структуры Package в строку при помощи его конвертации в JSON-формат. Используется функция MarshalIndent, которая вместе с упаковкой в формате JSON, ставит табуляции во вложенных его частях.

(1.4) Константа ENDBYTES.

```
const (
	ENDBYTES = "\000\005\007\001\001\007\005\000"
)
```

(1.5) Функция readPackage.

```
func readPackage(conn net.Conn) *Package {
	var (
		data   string
		size   = uint64(0)
		buffer = make([]byte, BUFFSIZE)
	)
	for {
		length, err := conn.Read(buffer)
		if err != nil {
			return nil
		}
		size += uint64(length)
		if size > DMAXSIZE {
			return nil
		}
		data += string(buffer[:length])
		if strings.Contains(data, ENDBYTES) {
			data = strings.Split(data, ENDBYTES)[0]
			break
		}
	}
	return DeserializePackage(data)
}
```

Читает данные с сокета conn до тех пор, пока не встретит константную строку завершения данных. Чтение происходит по порциям кратным константе BUFFSIZE (1.7). Если произошла ошибка чтения или количество общих переданных байт оказалось больше константы DMAXSIZE (1.8), тогда функция возвращает нулевой адрес. Если не произошло никакой ошибки, функция

возвращает объект типа Package, используя функцию DeserializePackage (1.9) (обратную к SerializePackage).

(1.6) Константа WAITTIME.

(1.7) Константа BUFFSIZE.

(1.8) Константа DMAXSIZE.

```
const (
	WAITTIME = 5 // seconds
	DMAXSIZE = (2 << 20) // (2^20)*2 = 2MiB
	BUFFSIZE = (4 << 10) // (2^10)*4 = 4KiB
)
```

(1.9) Функция DeserializePackage.

```
func DeserializePackage(data string) *Package {
	var pack Package
	err := json.Unmarshal([]byte(data), &pack)
	if err != nil {
		return nil
	}
	return &pack
}
```

Теперь осталось лишь написать функцию Listen (1.10), которая будет уже со стороны сервера (а точнее узла) прослушивать соединения. Она будет принимать в качестве адреса строку вида IPv4:Port и функцию-обработчик.

(1.10) Функция Listen.

```
func Listen(address string, handle func(Conn, *Package)) Listener {
```

```
        splited := strings.Split(address, ":")
        if len(splited) != 2 {
                return nil
        }
        listener, err := net.Listen("tcp", "0.0.0.0:"+splited[1])
        if err != nil {
                return nil
        }
        go serve(listener, handle)
        return Listener(listener)
}
```

Стоит заметить, что IPv4 адрес отбрасывается, так как в аргументах функции net.Listen нужен лишь порт, для принятия данных. При этом в качестве IPv4 используется шаблон (0.0.0.0), благодаря которому можно принимать соединения с разных адресов.

Прописанная же функция Listen запускает параллельную функцию serve (1.11), необходимую для обработки запросов, и возвращает сокет. Стоит заметить, что возвращаемый результат оборачивается не в функцию Listener, а преобразовывается в тип данных Listener (1.12). Это необходимо для того, чтобы программист, импортируя сделанный нами пакет, не подключал вместе с ним и стандартный пакет «net». Точно такая же ситуация происходит с типом Conn (1.13), в переданной функции handle.

(1.11) Функция serve.

```
func serve(listener net.Listener, handle func(Conn, *Package)) {
        defer listener.Close()
        for {
                conn, err := listener.Accept()
                if err != nil {
                        break
                }
                go handleConn(conn, handle)
```

```
        }
    }
    func handleConn(conn net.Conn, handle func(Conn, *Package)) {
        defer conn.Close()
        pack := readPackage(conn)
        if pack == nil {
            return
        }
        handle(Conn(conn), pack)
    }
```

Принимает сокет прослушивателя и функцию-обработчик. Предназначена для соединения с клиентом и последующего чтения данных (handleConn), посылаемых им же, с перенаправлением на функцию-обработчик handle. После завершения функции handle связь с клиентом будет автоматически прервана (defer conn.Close()).

(1.12) Тип данных Listener.

(1.13) Тип данных Conn.

```
type Listener net.Listener
type Conn net.Conn
```

Последнее, что необходимо сделать в библиотеке, это добавить функцию Handle (1.14). Она будет совершать действия исходя из опции принятого пакета.

(1.14) Функция Handle.

```
func Handle(option int, conn Conn, pack *Package, handle func(*Package) string) bool {
    if pack.Option != option {
        return false
```

```
        }
        conn.Write([]byte(SerializePackage(&Package{
                Option: option,
                Data:   handle(pack),
        }) + ENDBYTES))
        return true
}
```

Принимает в качестве аргументов: 1. опцию, на которую эта функция привязывает исполнение, 2. принимаемый пакет, от куда будут браться нужные данные, и 3. функцию, которая будет совершать действия исходя из данных, возвращая при этом результат проделанной работы в виде строки. Сама же функция Handle будет возвращать булево значение, исходя из совпадения опций.

Данная библиотека использует следующие пакеты:

```
import (
        "net"
        "strings"
        "time"
        "encoding/json"
)
```

Импорты вставляются после строки, указывающей имя пакета (package network), и перед всем остальным кодом.

Написав всю библиотеку, можно проверить корректность её исполнения на примере простой программы отправления-принятия данных. К тому же, этот код является неплохим шаблоном, который можно проецировать на другие приложения.

```
package main
```

```
import (
	"fmt"
	"time"
	"strings"
	nt "./network"
)
const (
	TO_UPPER = iota + 1
	TO_LOWER
)
const (
	ADDRESS = ":8080"
)
func main() {
	var (
		res = new(nt.Package)
		msg = "Hello, World!"
	)
	go nt.Listen(ADDRESS, handleServer)
	time.Sleep(500 * time.Millisecond)
	// send «Hello, World!»
	// receive «HELLO, WORLD!»
	res = nt.Send(ADDRESS, &nt.Package{
		Option: TO_UPPER,
		Data: msg,
	})
	fmt.Println(res.Data)
	// send «HELLO, WORLD!»
	// receive «hello, world!»
	res = nt.Send(ADDRESS, &nt.Package{
		Option: TO_LOWER,
		Data: res.Data,
	})
	fmt.Println(res.Data)
}
func handleServer(conn nt.Conn, pack *nt.Package) {
```

```
        nt.Handle(TO_UPPER, conn, pack, handleToUpper)
        nt.Handle(TO_LOWER, conn, pack, handleToLower)
}
func handleToUpper(pack *nt.Package) string {
        return strings.ToUpper(pack.Data)
}
func handleToLower(pack *nt.Package) string {
        return strings.ToLower(pack.Data)
}
```

В данном коде клиент отправляет серверу два последовательных запроса: сначала на перевод строки в верхний регистр, а потом в нижний. Также здесь есть такое ключевое слово как «iota», оно сигнализирует о нумерации констант (аналогично enum в языке Си). «iota+1» говорит о том, чтобы нумерация констант начиналась с единицы.

Назовём данный файл как «main.go». Он должен располагаться рядом с директорией «network/».

```
network/
        network.go
main.go
```

Чтобы скомпилировать и запустить этот код, необходимо прописать следующие команды в терминале:

```
go build main.go
./main
```

Первая команда - это компиляция, которая на основе исходного кода main.go создаёт исполняемый (машинный) код под файлом main. Вторая команда - исполнение.

Результат работы.

HELLO, WORLD!

hello, world!

3.2. Библиотека для работы с блокчейном

Библиотеку, работающую с блокчейном, назовём как «blockchain», а исходный файл под названием «blockchain.go» расположим в директории «blockchain/».

```
package blockchain
```

Приступать к написанию библиотеки необходимо с создания структур, на основе которых будут базироваться все последующие действия. Блокчейн даёт три основные структуры - BlockChain (2.1), Block (2.2), Transaction (2.3).

(2.1) Структура BlockChain.

```
type BlockChain struct {
	DB	*sql.DB
}
```

Хранит в себе указатель на базу данных, в которую добавляет и из которой берёт блоки.

(2.2) Структура Block.

```
type Block struct {
	Nonce		uint64
	Difficulty	uint8
	CurrHash	[]byte
	PrevHash	[]byte
	Transactions	[]Transaction
	Mapping		map[string]uint64
	Miner		string
	Signature	[]byte
```

```
    TimeStamp    string
}
```

Содержит следующие поля:

1. Difficulty (сложность блока) и Nonce (результат подтверждения работы);
2. CurrHash (хеш текущего блока) и PrevHash (хеш предыдущего блока);
3. Transactions (транзакции пользователей) и Mapping (состояния баланса пользователей);
4. Miner (пользователь замайнивший блок) и Signature (подпись майнера указывающая на CurrHash);
5. TimeStamp (метка времени создания блока);

CurrHash является результатом хеширования следующих значений: Difficulty, PrevHash, Transactions, Mapping, Miner и TimeStamp. Signature и Nonce привязаны к CurrHash.

(2.3) Структура Transaction.

```
type Transaction struct {
    RandBytes    []byte
    PrevBlock    []byte
    Sender       string
    Receiver     string
    Value        uint64
    ToStorage    uint64
    CurrHash     []byte
    Signature    []byte
}
```

Состоит из следующих полей:

1. RandBytes (случайные байты), PrevBlock (хеш последнего блока в блокчейне);

2. Sender (имя отправителя), Receiver (имя получателя), Value (количество переводимых денег получателю);
3. ToStorage (количество переводимых денег хранилищу);
4. CurrHash (хеш текущей транзакции), Signature (подпись отправителя);

CurrHash является результатом хеширования всех полей, кроме Signature.

Определив основные структуры данных, можно приступать к написанию функций. В первую очередь нужно реализовать функцию создания блокчейна NewChain (2.4).

(2.4) Функция NewChain.

```
func NewChain(filename, receiver string) error {
	file, err := os.Create(filename)
	if err != nil {
		return err
	}
	file.Close()
	db, err := sql.Open("sqlite3", filename)
	if err != nil {
		return err
	}
	defer db.Close()
	_, err = db.Exec(CREATE_TABLE)
	chain := &BlockChain{
		DB: db,
	}
	genesis := &Block{
		CurrHash:  []byte(GENESIS_BLOCK),
		Mapping:   make(map[string]uint64),
		Miner:     receiver,
		TimeStamp: time.Now().Format(time.RFC3339),
	}
```

```
        genesis.Mapping[STORAGE_CHAIN] = STORAGE_VALUE
        genesis.Mapping[receiver] = GENESIS_REWARD
        chain.AddBlock(genesis)
        return nil
}
```

Принимает в качестве первого аргумента имя файла, в качестве второго - имя пользователя в блокчейне (в контексте генезис-блока, имя майнера). Возвращает либо нулевой адрес как выполнение без ошибки, либо возвращает строку как обнаружение ошибки. Сначала функция создаёт файл при помощи функции os.Create, далее открывает этот файл как локальную БД, при помощи функции sql.Open. Исполняет команду создания таблицы, которая прописана в константе CREATE_TABLE (2.5). Далее создаётся переменная типа BlockChain, в которую заносится указатель на БД, и генезис-блок (2.6), в котором прописывается майнер, а также время создания. После этого в состояние генезис-блока заносятся балансы (2.7)(2.8) пользователей (хранилища (2.9) и создателя). И в конце, блок добавляется в БД при помощи метода AddBlock (2.10).

(2.5) Константа CREATE_TABLE.

```
const (
        CREATE_TABLE = `
CREATE TABLE BlockChain (
   Id INTEGER PRIMARY KEY AUTOINCREMENT,
   Hash VARCHAR(44) UNIQUE,
   Block TEXT
);
`
)
```

(2.6) Константа GENESIS_BLOCK.

(2.7) Константа STORAGE_VALUE.

(2.8) Константа GENESIS_REWARD.

(2.9) Константа STORAGE_CHAIN.

```
const (
	GENESIS_BLOCK  = "GENESIS-BLOCK"
	STORAGE_VALUE  = 100
	GENESIS_REWARD = 100
	STORAGE_CHAIN  = "STORAGE-CHAIN"
)
```

(2.10) Метод AddBlock.

```
func (chain *BlockChain) AddBlock(block *Block) {
	chain.DB.Exec("INSERT INTO BlockChain (Hash, Block) VALUES ($1, $2)",
		Base64Encode(block.CurrHash),
		SerializeBlock(block),
	)
}
```

Добавляет новый блок в БД. Использует функции Base64Encode (2.11) и SerializeBlock (2.12).

(2.11) Функция Base64Encode.

```
func Base64Encode(data []byte) string {
	return base64.StdEncoding.EncodeToString(data)
}
```

(2.12). Функция SerializeBlock.

```
func SerializeBlock(block *Block) string {
	jsonData, err := json.MarshalIndent(*block, "", "\t")
	if err != nil {
		return ""
	}
	return string(jsonData)
}
```

Чтобы воспользоваться уже созданным блокчейном нужно создать функцию подгрузки БД LoadChain (2.13).

(2.13) Функция LoadChain.

```
func LoadChain(filename string) *BlockChain {
	db, err := sql.Open("sqlite3", filename)
	if err != nil {
		return nil
	}
	chain := &BlockChain{
		DB: db,
	}
	return chain
}
```

Принимает в качестве аргумента имя файла и возвращает экземпляр структуры BlockChain. Открывает локальное БД.

Чтобы продолжать далее писать код связанный с блокчейном , необходимым является написание функций, связанных с действиями над блоками. Самой основной функцией является создание блока (2.14).

(2.14) Функция NewBlock.

```
func NewBlock(miner string, prevHash []byte) *Block {
	return &Block{
		Difficulty: DIFFICULTY,
		PrevHash:   prevHash,
		Miner:      miner,
		Mapping:    make(map[string]uint64),
	}
}
```

Создаёт шаблон блока. Принимает в качестве аргументов имя майнера и хеш предыдущего блока. Использует также константу DIFFICULTY (2.15). Константность сложности обусловлена лёгкостью воспроизведения кода, без оглядки на саморегулирование сложности цепи.

(2.15) Константа DIFFICULTY.

```
const (
	DIFFICULTY = 20
)
```

Значение константы равно количеству первых нулевых бит в хеше.

После создания блока, в него нужно вносить транзакции пользователей. Чтобы создать транзакцию, надлежит использовать функцию NewTransaction (2.16).

(2.16) Функция NewTransaction.

```
func NewTransaction(user *User, lasthash []byte, to string, value uint64) *Transaction {
	tx := &Transaction{
		RandBytes: GenerateRandomBytes(RAND_BYTES),
		PrevBlock: lasthash,
		Sender:    user.Address(),
```

```
            Receiver:  to,
            Value:     value,
        }
        if value > START_PERCENT {
            tx.ToStorage = STORAGE_REWARD
        }
        tx.CurrHash = tx.hash()
        tx.Signature = tx.sign(user.Private())
        return tx
}
```

Принимает в качестве аргументов объект структуры User (2.17), хеш последнего блока в блокчейне, имя получателя и количество отправляемых средств. Сама функция создаёт транзакцию, проверяет количество отправляемых средств и если они превышают константу START_PERCENT, тогда из кошелька берутся дополнительные деньги в размере STORAGE_REWARD, которые перенаправятся в хранилище. В данном коде появляются новые элементы, ранее неизвестные, такие как функция GenerateRandomBytes (2.18), константы RAND_BYTES (2.19), START_PERCENT (2.20), STORAGE_REWARD (2.21) и методы Address (2.22), Private (2.23), hash (2.24), sign (2.25).

(2.17) Структура User.

```
type User struct {
    PrivateKey *rsa.PrivateKey
}
```

Представляет из себя лишь оболочку над типом данных rsa.PrivateKey.

(2.18) Функция GenerateRandomBytes.

```
func GenerateRandomBytes(max uint) []byte {
```

```
        var slice []byte = make([]byte, max)
        _, err := rand.Read(slice)
        if err != nil {
                return nil
        }
        return slice
}
```

Принимает в качестве аргумента число, как количество необходимых байт для генерации. Возвращает срез из псевдослучайных байт.

(2.19) Константа RAND_BYTES.

(2.20) Константа START_PERCENT.

(2.21) Константа STORAGE_REWARD.

```
const (
        RAND_BYTES = 32
        START_PERCENT = 10
        STORAGE_REWARD = 1
)
```

(2.22) Метод Address.

```
func (user *User) Address() string {
        return StringPublic(user.Public())
}
```

Преобразует публичный ключ в строку. Использует функцию StringPublic (2.26) и метод Public (2.27).

(2.23) Метод Private.

```
func (user *User) Private() *rsa.PrivateKey {
	return user.PrivateKey
}
```

(2.24) Метод hash.

```
func (tx *Transaction) hash() []byte {
	return HashSum(bytes.Join(
		[][]byte{
			tx.RandBytes,
			tx.PrevBlock,
			[]byte(tx.Sender),
			[]byte(tx.Receiver),
			ToBytes(tx.Value),
			ToBytes(tx.ToStorage),
		},
		[]byte{},
	))
}
```

Конкатенирует байты полей объекта, после чего производит над получившимся значением хеширование. Использует функции HashSum (2.28) и ToBytes (2.29).

(2.25) Метод sign. Использует функцию Sign (2.30).

```
func (tx *Transaction) sign(priv *rsa.PrivateKey) []byte {
	return Sign(priv, tx.CurrHash)
}
```

(2.26) Функция StringPublic.

```
func StringPublic(pub *rsa.PublicKey) string {
	return Base64Encode(x509.MarshalPKCS1PublicKey(pub))
}
```

Переводит публичный ключ в набор байтов, после чего применяет кодировку base64 для трансляции в строку.

(2.27) Метод Public.

```
func (user *User) Public() *rsa.PublicKey {
	return &(user.PrivateKey).PublicKey
}
```

(2.28) Функция HashSum.

```
func HashSum(data []byte) []byte {
	hash := sha256.Sum256(data)
	return hash[:]
}
```

(2.29) Функция ToBytes.

```
func ToBytes(num uint64) []byte {
	var data = new(bytes.Buffer)
	err := binary.Write(data, binary.BigEndian, num)
	if err != nil {
		return nil
	}
	return data.Bytes()
}
```

Переводит число в набор байтов.

(2.30) Функция Sign.

```
func Sign(priv *rsa.PrivateKey, data []byte) []byte {
	signature, err := rsa.SignPSS(rand.Reader, priv, crypto.SHA256, data, nil)
	if err != nil {
		return nil
	}
	return signature
}
```

Подписывает данные исходя из приватного ключа.

После создания транзакции её необходимо добавить в блок. Для этого создадим метод AddTransaction (2.31).

(2.31) Метод AddTransaction.

```
func (block *Block) AddTransaction(chain *BlockChain, tx *Transaction) error {
	if tx == nil {
		return errors.New("tx is null")
	}
	if tx.Value == 0 {
		return errors.New("tx value = 0")
	}
	if tx.Sender != STORAGE_CHAIN && len(block.Transactions) == TXS_LIMIT {
		return errors.New("len tx = limit")
	}
	if tx.Sender != STORAGE_CHAIN && tx.Value > START_PERCENT &&
tx.ToStorage != STORAGE_REWARD {
		return errors.New("storage reward pass")
	}
	if !bytes.Equal(tx.PrevBlock, chain.LastHash()) {
		return errors.New("prev block in tx /= last hash in chain")
```

```
        }
        var balanceInChain uint64
        balanceInTX := tx.Value + tx.ToStorage
        if value, ok := block.Mapping[tx.Sender]; ok {
                balanceInChain = value
        } else {
                balanceInChain = chain.Balance(tx.Sender, chain.Size())
        }
        if balanceInTX > balanceInChain {
                return errors.New("insufficient funds")
        }
        block.Mapping[tx.Sender] = balanceInChain - balanceInTX
        block.addBalance(chain, tx.Receiver, tx.Value)
        block.addBalance(chain, STORAGE_CHAIN, tx.ToStorage)
        block.Transactions = append(block.Transactions, *tx)
        return nil
}
```

Метод привязан к блоку, принимает в качестве аргументов объект типа BlockChain и объект типа Transaction. Возвращает нулевой адрес, как результат выполнения без ошибки, либо строку, как результат выполнения с ошибкой. Данная функция сначала сравнивает транзакцию и отправляемые средства на нуль, далее проверяет при помощи константы TXS_LIMIT (2.32) перегруженность блока транзакциями (блок имеет фиксированное количество допустимых транзакций). После чего берётся баланс отправителя, при помощи метода Balance (2.33) или при помощи состояния, если в блоке уже существуют транзакции от этого пользователя. Далее идёт сравнение отправляемых данных с константами START_PERCENT, STORAGE_REWARD. И последнее сравнение связано с указанным балансом в транзакции и общим балансом пользователя по блокчейну. Если проверки оказались успешными (то-есть без ошибок), тогда состояние отправителя обновляется, а после и состояния получателя вместе с хранилищем (2.34). В конце, данная транзакция заносится в список транзакций блока. Используется метод Size (2.35).

(2.32) Константа TXS_LIMIT.

```
const (
	TXS_LIMIT = 2
)
```

Определяет максимальное количество транзакций в одном блоке.

(2.33) Метод Balance.

```
func (chain *BlockChain) Balance(address string, size uint64) uint64 {
	var (
		sblock  string
		block   *Block
		balance uint64
	)
	rows, err := chain.DB.Query("SELECT Block FROM BlockChain WHERE Id <= $1
ORDER BY Id DESC", size)
	if err != nil {
		return balance
	}
	defer rows.Close()
	for rows.Next() {
		rows.Scan(&sblock)
		block = DeserializeBlock(sblock)
		if value, ok := block.Mapping[address]; ok {
			balance = value
			break
		}
	}
	return balance
}
```

Привязан к объекту структуры BlockChain. В качестве аргумента принимает имя пользователя. Возвращает баланс указанного пользователя. Производит считывание блоков с конца цепочки, пока не найдёт состояние, в котором имя пользователя будет указано. После нахождения весь дальнейший поиск прекращается. При взятии блока из локальной БД используется функция DeserializeBlock (2.36).

(2.34) Метод addBalance.

```
func (block *Block) addBalance(chain *BlockChain, receiver string, value uint64) {
	var balanceInChain uint64
	if v, ok := block.Mapping[receiver]; ok {
		balanceInChain = v
	} else {
		balanceInChain = chain.Balance(receiver, chain.Size())
	}
	block.Mapping[receiver] = balanceInChain + value
}
```

Привязан к объекту структуры Block. Принимает в качестве аргументов объект типа BlockChain, получателя и переданные ему средства, после чего производит обновление состояния.

(2.35) Метод Size.

```
func (chain *BlockChain) Size() uint64 {
	var size uint64
	row := chain.DB.QueryRow("SELECT Id FROM BlockChain ORDER BY Id DESC")
	row.Scan(&size)
	return size
}
```

Возвращает количество блоков в локальной БД.

(2.36) Функция DeserializeBlock.

```
func DeserializeBlock(data string) *Block {
	var block Block
	err := json.Unmarshal([]byte(data), &block)
	if err != nil {
		return nil
	}
	return &block
}
```

Поместив все транзакции в блок, его необходимо подтвердить во-первых подписью, во-вторых работой (PoW). Для этого создадим метод Accept (2.37).

(2.37) Метод Accept.

```
func (block *Block) Accept(chain *BlockChain, user *User, ch chan bool) error {
	if !block.transactionsIsValid(chain, chain.Size()) {
		return errors.New("transactions is not valid")
	}
	block.AddTransaction(chain, &Transaction{
		RandBytes: GenerateRandomBytes(RAND_BYTES),
		PrevBlock: chain.LastHash(),
		Sender:    STORAGE_CHAIN,
		Receiver:  user.Address(),
		Value:     STORAGE_REWARD,
	})
	block.TimeStamp = time.Now().Format(time.RFC3339)
	block.CurrHash = block.hash()
	block.Signature = block.sign(user.Private())
	block.Nonce = block.proof(ch)
```

```
        return nil
    }
```

Привязан к объекту структуры Block. Принимает в качестве аргументов объект типа BlockChain, объект типа User и канал типа chan bool. Возвращает нулевой адрес при успешном выполнении, либо строку, при обнаружении ошибки. Канал (chan bool) необходим для прекращения выполнения PoW. Это необходимо в моменты, когда другой узел найдёт подходящий хеш быстрее, соответственно и замайнитсам блок, из чего следует, что нет смысла продолжать дальше пытаться майнить этот хеш. Сама функция использует метод transactionsIsValid (2.38) для проверки правильности транзакций, которые находятся в блоке. После чего использует функцию добавления транзакции от имени хранилища, переводя награду майнеру. Если в хранилище не останется средств, тогда метод AddTransaction выдаст ошибку и не добавит транзакцию в блок. Далее на блок помещается метка времени (следует заметить, что метка использует именно локальное время), вычисляется хеш (2.39), блок подписывается (2.40) и подтверждается работой (2.41).

(2.38) Метод transactionsIsValid.

```
func (block *Block) transactionsIsValid(chain *BlockChain, size uint64) bool {
        lentxs := len(block.Transactions)
        plusStorage := 0
        for i := 0; i < lentxs; i++ {
                if block.Transactions[i].Sender == STORAGE_CHAIN {
                        plusStorage = 1
                        break
                }
        }
        if lentxs == 0 || lentxs > TXS_LIMIT+plusStorage {
                return false
        }
        for i := 0; i < lentxs-1; i++ {
```

```
                for j := i + 1; j < lentxs; j++ {
                        if bytes.Equal(block.Transactions[i].RandBytes,
block.Transactions[j].RandBytes) {
                                return false
                        }
                        if      block.Transactions[i].Sender == STORAGE_CHAIN &&
                                block.Transactions[j].Sender == STORAGE_CHAIN {
                                        return false
                        }
                }
        }
        for i := 0; i < lentxs; i++ {
                tx := block.Transactions[i]
                if tx.Sender == STORAGE_CHAIN {
                        if tx.Receiver != block.Miner || tx.Value != STORAGE_REWARD {
                                return false
                        }
                } else {
                        if !tx.hashIsValid() {
                                return false
                        }
                        if !tx.signIsValid() {
                                return false
                        }
                }
                if !block.balanceIsValid(chain, tx.Sender, size) {
                        return false
                }
                if !block.balanceIsValid(chain, tx.Receiver, size) {
                        return false
                }
        }
        return true
}
```

Привязан к объекту структуры Block. В качестве аргумента принимает объект типа BlockChain. Возвращает true, если все транзакции валидны, иначе false. Сначала сравнивает лимит транзакций блока с количеством транзакций в блоке. Далее проверяет не существует ли одинаковых случайных байт и не существует ли больше чем одной транзакции от хранилища (в одном блоке может существовать лишь одна транзакция, где отправителем является само хранилище). Далее проверяется каждая отдельная транзакция, проверяется её хеш (2.42), подпись отправителя (2.43), и сравниваются балансы отправителя и получателя по состоянию блока при помощи метода balanceIsValid (2.44).

(2.39) Метод hash.

```
func (block *Block) hash() []byte {
	var tempHash []byte
	for _, tx := range block.Transactions {
		tempHash = HashSum(bytes.Join(
			[][]byte{
				tempHash,
				tx.CurrHash,
			},
			[]byte{},
		))
	}
	var list []string
	for hash := range block.Mapping {
		list = append(list, hash)
	}
	sort.Strings(list)
	for _, hash := range list {
		tempHash = HashSum(bytes.Join(
			[][]byte{
				tempHash,
				[]byte(hash),
				ToBytes(block.Mapping[hash]),
```

```
                },
                []byte{},
            ))
        }
        return HashSum(bytes.Join(
            [][]byte{
                tempHash,
                ToBytes(uint64(block.Difficulty)),
                block.PrevHash,
                []byte(block.Miner),
                []byte(block.TimeStamp),
            },
            []byte{},
        ))
    }
```

Сначала поочерёдно хешируются все транзакции. Далее осуществляется перевод хеш-таблицы в срез, после чего производится сортировка имён пользователе (это необходимо делать, так как запись и последующее хранение в хеш-таблицах могут отличаться друг от друга, тем самым приводя к получению разных хешей). После сортировки, поочерёдно начинает хешироваться значения из среза. И в конце, осуществляется хеширование всех других полей в совокупности с транзакциями и состояниями.

(2.40) Метод sign.

```
func (block *Block) sign(priv *rsa.PrivateKey) []byte {
    return Sign(priv, block.CurrHash)
}
```

(2.41) Метод proof. Использует функцию ProofOfWork (2.45).

```
func (block *Block) proof(ch chan bool) uint64 {
```

```
        return ProofOfWork(block.CurrHash, block.Difficulty, ch)
}
```

(2.42) Метод hashIsValid.

```
func (tx *Transaction) hashIsValid() bool {
        return bytes.Equal(tx.hash(), tx.CurrHash)
}
```

(2.43) Метод signIsValid.

```
func (tx *Transaction) signIsValid() bool {
        return Verify(ParsePublic(tx.Sender), tx.CurrHash, tx.Signature) == nil
}
```

Переводит имя пользователя (которое является лишь публичным ключом в строковом формате) в тип *rsa.PublicKey. Далее с этим ключом проверяется правильность подписанного хеша. Использует функции Verify (2.46) и ParsePublic (2.47).

(2.44) Метод balanceIsValid.

```
func (block *Block) balanceIsValid(chain *BlockChain, address string, size uint64) bool {
        if _, ok := block.Mapping[address]; !ok {
                return false
        }
        lentxs := len(block.Transactions)
        balanceInChain := chain.Balance(address, size)
        balanceSubBlock := uint64(0)
        balanceAddBlock := uint64(0)
        for j := 0; j < lentxs; j++ {
```

```
                tx := block.Transactions[j]
                if tx.Sender == address {
                        balanceSubBlock += tx.Value + tx.ToStorage
                }
                if tx.Receiver == address {
                        balanceAddBlock += tx.Value
                }
                if STORAGE_CHAIN == address {
                        balanceAddBlock += tx.ToStorage
                }
        }
        if (balanceInChain + balanceAddBlock - balanceSubBlock) !=
block.Mapping[address] {
                return false
        }
        return true
}
```

Проверяет совместимость данных, которые хранятся в транзакциях, с данными, которые хранятся в состоянии по указанному имени пользователи. Если существует какое-то различие, тогда возвращается значение false, как результат того, что данные невалидны, иначе true. Правильность вычисляется по формуле: balanceInChain + balanceAddBlock - balanceSubBlock. То-есть сначала берётся баланс, хранимый в блокчейне, к нему прибавляются средства полученные и вычитаются средства отправленные. Если полученное значение не равняется значению, хранимому в состоянии, тогда выдаётся ошибка.

(2.45) Функция ProofOfWork.

```
func ProofOfWork(blockHash []byte, difficulty uint8, ch chan bool) uint64 {
        var (
                Target  = big.NewInt(1)
                intHash = big.NewInt(1)
                nonce   = uint64(mrand.Intn(math.MaxUint32))
```

```
		hash	[]byte
	)
	Target.Lsh(Target, 256 - uint(difficulty))
	for nonce < math.MaxUint64 {
		select {
		case <-ch:
			if DEBUG {
				fmt.Println()
			}
			return nonce
		default:
			hash = HashSum(bytes.Join(
				[][]byte{
					blockHash,
					ToBytes(nonce),
				},
				[]byte{},
			))
			if DEBUG {
				fmt.Printf("\rMining: %s", Base64Encode(hash))
			}
			intHash.SetBytes(hash)
			if intHash.Cmp(Target) == -1 {
				if DEBUG {
					fmt.Println()
				}
				return nonce
			}
			nonce++
		}
	}
	return nonce
}
```

Создаёт локальную переменную Target, которая создана для сравнения с получаемым хешем. Изначально Target равняется единице, что единице в

двоичном виде. Далее производится сдвиг влево на (256 - DIFFICULTY) битов. Чем меньше DIFFICULTY, тем легче найти значение хеша. В nonce устанавливается случайное число, в диапазоне [0;UINT32_MAX), во избежание одних и тех же вычислений со стороны разных узлов (так как в данной реализации не существует пул-серверов). Каждый раз nonce инкрементируется на единицу, после чего конкатенируется с хешем блока и хешируется. Полученный хеш преобразуется в число, после чего сравнивается с Target. И если полученное число оказывается меньше Target'а, это говорит о том, что работа подтверждена (нашёлся такой хеш, количество начальных битовых нулей которого равно или больше указанного числа в DIFFICULTY). Использует константу DEBUG (2.48). При этом стоит заметить, что число nonce генерируется исходя из функции Intn (пакет math/rand), которое в свою очередь использует seed при генерации псевдослучайной последовательности. По-умолчанию seed является статичным и не изменяется, тем самым при запуске приложения генерируемая последовательность будет всегда одинаковой. Чтобы избавиться от этого, необходимо при каждом запуске приложения изменять seed. В качестве seed'а можно ложить текущее время (2.49).

(2.46) Функция Verify.

```
func Verify(pub *rsa.PublicKey, data, sign []byte) error {
	return rsa.VerifyPSS(pub, crypto.SHA256, data, sign, nil)
}
```

Использует публичный ключ для проверки подписанных данных с первоначальными.

(2.47) Функция ParsePublic.

```
func ParsePublic(pubData string) *rsa.PublicKey {
	pub, err := x509.ParsePKCS1PublicKey(Base64Decode(pubData))
	if err != nil {
```

```
            return nil
        }
        return pub
    }
```

Использует функцию Base64Decode (2.50).

(2.48) Константа DEBUG.

```
    const (
        DEBUG = true
    )
```

(2.49) Функция init.

```
    func init() {
        mrand.Seed(time.Now().UnixNano())
    }
```

Изменяет seed генератора псевдослучайных чисел, из пакета math/rand, методом вычисления текущего времени. Запустится автоматически, при подключении пакета «blockchain».

(2.50) Функция Base64Decode.

```
    func Base64Decode(data string) []byte {
        result, err := base64.StdEncoding.DecodeString(data)
        if err != nil {
            return nil
        }
        return result
    }
```

Осталось лишь реализовать дополнительные функции и методы: генерация приватного ключа GeneratePrivate (2.51), перевод приватного ключа в строку StringPrivate (2.52), перевод строки в приватный ключ ParsePrivate (2.53), создать пользователя NewUser (2.54), загрузить пользователя LoadUser (2.55), получить кошелёк Purse (2.56), получить последний хеш в блокчейне LastHash (2.57), проверить блок IsValid (2.58), сериализовать транзакцию SerializeTX (2.59), десериализовать транзакцию DeserializeTX (2.60).

(2.51) Функция GeneratePrivate.

```
func GeneratePrivate(bits uint) *rsa.PrivateKey {
	priv, err := rsa.GenerateKey(rand.Reader, int(bits))
	if err != nil {
		return nil
	}
	return priv
}
```

(2.52) Функция StringPrivate.

```
func StringPrivate(priv *rsa.PrivateKey) string {
	return Base64Encode(x509.MarshalPKCS1PrivateKey(priv))
}
```

(2.53) Функция ParsePrivate.

```
func ParsePrivate(privData string) *rsa.PrivateKey {
	priv , err := x509.ParsePKCS1PrivateKey(Base64Decode(privData))
	if err != nil {
```

```
                return nil
        }
        return priv
}
```

(2.54) Функция NewUser.

```
func NewUser() *User {
        return &User{
                PrivateKey: GeneratePrivate(KEY_SIZE),
        }
}
```

Использует константу KEY_SIZE (2.61).

(2.55) Функция LoadUser.

```
func LoadUser(purse string) *User {
        priv := ParsePrivate(purse)
        if priv == nil {
                return nil
        }
        return &User{
                PrivateKey: priv,
        }
}
```

(2.56) Метод Purse.

```
func (user *User) Purse() string {
        return StringPrivate(user.Private())
}
```

(2.57) Метод LastHash.

```
func (chain *BlockChain) LastHash() []byte {
	var hash string
	row := chain.DB.QueryRow("SELECT Hash FROM BlockChain ORDER BY Id DESC")
	row.Scan(&hash)
	return Base64Decode(hash)
}
```

(2.58) Метод IsValid.

```
func (block *Block) IsValid(chain *BlockChain, size uint64) bool {
	switch {
	case block == nil:
		return false
	case block.Difficulty != DIFFICULTY:
		return false
	case !block.hashIsValid(chain, size):
		return false
	case !block.signIsValid():
		return false
	case !block.proofIsValid():
		return false
	case !block.mappingIsValid():
		return false
	case !block.timeIsValid(chain):
		return false
	case !block.transactionsIsValid(chain, size):
		return false
	}
	return true
```

```
}
```

Использует методы hashIsValid (2.62), signIsValid (2.63), proofIsValid (2.64), mappingIsValid (2.65), timeIsValid (2.66).

(2.59) Функция SerializeTX.

```
func SerializeTX(tx *Transaction) string {
	jsonData, err := json.MarshalIndent(*tx, "", "\t")
	if err != nil {
		return ""
	}
	return string(jsonData)
}
```

(2.60) Функция DeserializeTX.

```
func DeserializeTX(data string) *Transaction {
	var tx Transaction
	err := json.Unmarshal([]byte(data), &tx)
	if err != nil {
		return nil
	}
	return &tx
}
```

(2.61) Константа KEY_SIZE.

```
const (
	KEY_SIZE = 512
)
```

(2.62) Метод hashIsValid.

```
func (block *Block) hashIsValid(chain *BlockChain, size uint64) bool {
        if !bytes.Equal(block.hash(), block.CurrHash) {
                return false
        }
        var id uint64
        row := chain.DB.QueryRow("SELECT Id FROM BlockChain WHERE Hash=$1",
                Base64Encode(block.PrevHash))
        row.Scan(&id)
        return id == size
}
```

Проверяет хеш текущего блока пропущенного через метод hash с хешем, хранимым в поле блоке. Также проверяет хеш предыдущего блока из блокчейна с хешем, хранимым в поле блока методом получения его ID из БД. Возвращает false при обнаружении ошибки, иначе true.

(2.63) Метод signIsValid.

```
func (block *Block) signIsValid() bool {
        return Verify(ParsePublic(block.Miner), block.CurrHash, block.Signature) == nil
}
```

(2.64) Метод proofIsValid.

```
func (block *Block) proofIsValid() bool {
        intHash := big.NewInt(1)
        Target := big.NewInt(1)
        hash := HashSum(bytes.Join(
                [][]byte{
                        block.CurrHash,
```

```
                ToBytes(block.Nonce),
            },
            []byte{},
        ))
        intHash.SetBytes(hash)
        Target.Lsh(Target, 256 - uint(block.Difficulty))
        if intHash.Cmp(Target) == -1 {
            return true
        }
        return false
}
```

Проверяет правильность работы используя указанные в блоке поля Nonce и CurrHash со сложностью Difficulty. Возвращает false, если работа не была выполнена, иначе true.

(2.65) Метод mappingIsValid.

```
func (block *Block) mappingIsValid() bool {
    for hash := range block.Mapping {
        if hash == STORAGE_CHAIN {
            continue
        }
        flag := false
        for _, tx := range block.Transactions {
            if tx.Sender == hash || tx.Receiver == hash {
                flag = true
                break
            }
        }
        if !flag {
            return false
        }
    }
    return true
```

```
}
```

Проверяет состояние блока на наличие пользователей, которые не указаны в транзакциях. Если таковые имеются, тогда функция возвращает false, как результат обнаружения ошибки, иначе true. Исключением является лишь хранилище, так как ему передаются средства через поле ToStorage, неявной транзакцией.

(2.66) Метод timeIsValid.

```
func (block *Block) timeIsValid(chain *BlockChain) bool {
	btime, err := time.Parse(time.RFC3339, block.TimeStamp)
	if err != nil {
		return false
	}
	diff := time.Now().Sub(btime)
	if diff < 0 {
		return false
	}
	var sblock string
	row := chain.DB.QueryRow("SELECT Block FROM BlockChain WHERE
Hash=$1",
		Base64Encode(block.PrevHash))
	row.Scan(&sblock)
	lblock := DeserializeBlock(sblock)
	if lblock == nil {
		return false
	}
	ltime, err := time.Parse(time.RFC3339, lblock.TimeStamp)
	if err != nil {
		return false
	}
	result := btime.Sub(ltime)
	return result > 0
}
```

Первым шагом проверяется время, указанное в блоке, с текущим времен. Если указанное время в блоке больше текущего, тогда возвращается false, как результат ошибки. Далее проверяется время указанное в предыдущем блоке. Если время предыдущего блока больше времени текущего блока, тогда также возвращается false как результат с ошибкой, иначе true.

Данная библиотека использует следующие пакеты:

```
import (
	"time"
	"errors"
	"bytes"
	"sort"
	"database/sql"
	_ "github.com/mattn/go-sqlite3"
	"os"
	"crypto"
	"crypto/rand"
	"crypto/rsa"
	"crypto/sha256"
	"crypto/x509"
	"encoding/base64"
	"encoding/binary"
	"encoding/json"
	"fmt"
	"math"
	"math/big"
	mrand "math/rand"
)
```

Написав всю библиотеку, можно проверить корректность её исполнения на примере простой программы создания блоков. Этот код также будет

являться неплохим шаблоном для последующего его проецирования на приложение узла.

```
package main
import (
	"fmt"
	bc "./blockchain"
)
const (
	DBNAME = "blockchain.db"
)
func main() {
	miner := bc.NewUser()
	bc.NewChain(DBNAME, miner.Address())
	chain := bc.LoadChain(DBNAME)
	for i := 0; i < 3; i++ {
		block := bc.NewBlock(miner.Address(), chain.LastHash())
		block.AddTransaction(chain,
			bc.NewTransaction(miner, chain.LastHash(), "aaa", 5))
		block.AddTransaction(chain,
			bc.NewTransaction(miner, chain.LastHash(), "bbb", 3))
		block.Accept(chain, miner, make(chan bool))
		chain.AddBlock(block)
	}
	var sblock string
	rows, err := chain.DB.Query("SELECT Block FROM BlockChain")
	if err != nil {
		panic("error: query to db")
	}
	for rows.Next() {
		rows.Scan(&sblock)
		fmt.Println(sblock)
	}
}
```

В данном коде создаётся майнер, при помощи функции NewUser, далее он же указывается создателем блокчейна. Имея средства, он перенаправляет их на другие адреса («aaa», «bbb») методом создания блока и занесения своих транзакций в этот блок, после чего подтверждает блок и заносит его в блокчейн. В конце программы у БД запрашиваются все блоки, которые она хранит, после чего осуществляется их вывод.

Назовём данный файл как «main.go». Он должен располагаться рядом с директорией «blockchain/».

```
blockchain/
        blockchain.go
main.go
```

Чтобы скомпилировать и запустить этот код, необходимо прописать следующие команды в терминале:

```
go build main.go
./main
```

Результат работы (сам результат всегда будет различаться за счёт использования разного времени и псевдослучайных данных, но структура останется той же).

```
Mining: AAAO5I2yq5bVdlgc2fwTRt6EHyNtGx83wkknMTpssrI=
Mining: AAAJ5w65nEn2chP0+82Q95wpZGKvkSXtSJhvlNK6914=
Mining: AAABVxEyH8HP5ERLWTgJn0VFw2kpckNZI03nFW7lpnY=
...
{
        "Nonce": 1958934804,
        "Difficulty": 20,
```

```
        "CurrHash": "qiZqMnhrAJqDK+iPdwc1O1sfKH0r9JgrzYLuolfphRQ=",
        "PrevHash": "kckTOJ5tyLiDT+2bCkdk+exmFgRsXvZHo45FQTAPVmg=",
        "Transactions": [
                {
                        "RandBytes":
"Yn6OG6U4EEH5HRntgq39hfwgYKiAwk7rqg3M0D30TYs=",
                        "PrevBlock": "kckTOJ5tyLiDT+2bCkdk+exmFgRsXvZHo45FQTAPVmg=",
                        "Sender":
"MEgCQQDC7/4owVPVNI824jP4rQgtRSt4+GXTJXv99l6ue2LHQa9FbFbc7jTV7ipmJtRJ1kgcZ
Novx1ZH12w1QUdt70rpAgMBAAE=",
                        "Receiver": "aaa",
                        "Value": 5,
                        "ToStorage": 0,
                        "CurrHash": "+RT60qjzzpFNe+YHH5mR2WjNotLZLEBVkT5ajJl91AI=",
                        "Signature":
"hMW4fy0kGA9dDmOE7lq68BACsqJDUBqUMADay23R3E8jqmQpsfQ0wS94INPNDMmTk6g
FF58sT7ycXbDC3wWcgQ=="
                },
                {
                        "RandBytes": "OvqFb8DEVMMFo0CS4PL91txNG7Y1Co+jojip+n73OS8=",
                        "PrevBlock": "kckTOJ5tyLiDT+2bCkdk+exmFgRsXvZHo45FQTAPVmg=",
                        "Sender":
"MEgCQQDC7/4owVPVNI824jP4rQgtRSt4+GXTJXv99l6ue2LHQa9FbFbc7jTV7ipmJtRJ1kgcZ
Novx1ZH12w1QUdt70rpAgMBAAE=",
                        "Receiver": "bbb",
                        "Value": 3,
                        "ToStorage": 0,
                        "CurrHash": "3FDpgkXHehrqp/fyvNHge6H8qzbPuQaJzNXKfDuA734=",
                        "Signature":
"lr9UJ2d5NBjw/m4Vv6541xkFt/HTFK2PzKJflFSPO1JbSbBF1QqjW8Opa6U8HeXt4lHb+Bq3uxP
xHEWrUj0L+g=="
                },
                {
                        "RandBytes": "8slLSjJzWDPbLuAr6I/FdH0e24kVOq7ZyftAOgBarFE=",
                        "PrevBlock": null,
                        "Sender": "STORAGE-CHAIN",
```

```
                "Receiver":
"MEgCQQDC7/4owVPVNI824jP4rQgtRSt4+GXTJXv99l6ue2LHQa9FbFbc7jTV7ipmJtRJ1kgcZ
Novx1ZH12w1QUdt70rpAgMBAAE=",
                "Value": 1,
                "ToStorage": 0,
                "CurrHash": null,
                "Signature": null
            }
        ],
        "Mapping": {
        "MEgCQQDC7/4owVPVNI824jP4rQgtRSt4+GXTJXv99l6ue2LHQa9FbFbc7jTV7ipmJtRJ
1kgcZNovx1ZH12w1QUdt70rpAgMBAAE=": 79,
            "STORAGE-CHAIN": 97,
            "aaa": 15,
            "bbb": 9
        },
        "Miner":
"MEgCQQDC7/4owVPVNI824jP4rQgtRSt4+GXTJXv99l6ue2LHQa9FbFbc7jTV7ipmJtRJ1kgcZ
Novx1ZH12w1QUdt70rpAgMBAAE=",
        "Signature":
"fXcaZH/0jUV6eRKgnxWOXpQuJxw5ZB9B5fU8gHYybwamZ0srwEkC2TZadkgfPDrcBcVtfaA3
ndlS29vH/RhLwQ==",
        "TimeStamp": "2020-07-17T06:38:39-04:00"
}
```

В результате работы программы показан момент с майнингом, а также был сокращён весь список блоков до последнего. В нём можно увидеть последнее состояние пользователей. Так например, майнер перевёл пользователям «aaa» и «bbb» в совокупности 24 монеты, и у самого майнера их осталось 79. При этом, у майнера до всех этих транзакций было 100 монет. 3 монеты взялись из хранилища и передались майнеру за проделанную работу. В итоге, хранилище имеет 97 монет в своей «казне».

При этом, если бы в одной из транзакций было указано больше 10 монет, тогда у хранилища запас «казны» пополнился бы на одну монету, за счёт её

взятия у майнера отправляющего свои монеты. В итоге, баланс хранилища оставался бы постоянным (равным 100), а майнер, переводящий монеты, выплачивал эту комиссию из майнинга (за счёт того, что в реализации, награда за майнинг равна комиссии за транзакцию, при сумме большей 10).

3.3. Клиент

Клиентское приложение будет состоять из двух частей:

1. Инициализация, при которой необходимо указать файл на приватный ключ пользователя (либо создать такой ключ), а также на адрес узлов.
2. Консольный интерфейс, который будет реализован через стандартный поток ввода (stdin). В нём можно будет прописывать команды проверки баланса, отправления транзакций и вывода цепочки блоков.

Стоит начать с инициализации. Примером входа в программу должна служить следующая строка:

./client -loaduser:private.key -loadaddr:addrlist.json

Указывается подгружаемый приватный ключ, который создаст объект структуры User, а также подгружается список адресов. Стоит заметить, что список адресов должен иметь формат json. Помимо параметра подгрузки пользователя (-loaduser:) можно использовать параметр создания нового пользователя (-newuser:). Если приватный ключ уже создан и хранится в файле, а в параметре newuser указан этот же файл, тогда произойдёт перезапись существующего файла, создав тем самым новый приватный ключ.

Пример подгружаемого файла для параметра loaduser, указывающего на адреса localhost (так как поля до ‘:’ пусты) с портами 8080, 9090:

```
[
    ":8080",
    ":9090"
]
```

Если какой-либо один из файлов не был указан или в файле была допущена ошибка (невалидный приватный ключ, неправильный синтаксис JSON формата), тогда программа завершится с ошибкой.

Также программа завершится с ошибкой, если список адресов будет пустым (характерно только для клиентского приложения).

Так как клиент является исполняемым приложением, его пакет именуется как «main».

```
package main
```

(3.1) Функция init.

```
func init() {
	if len(os.Args) < 2 {
		panic("failed: len(os.Args) < 2")
	}
	var (
		addrStr = ""
		userNewStr = ""
		userLoadStr = ""
	)
	var (
		addrExist = false
		userNewExist = false
		userLoadExist = false
	)
	for i := 1; i < len(os.Args); i++ {
		arg := os.Args[i]
		switch {
		case strings.HasPrefix(arg, "-loadaddr:"):
			addrStr = strings.Replace(arg, "-loadaddr:", "", 1)
			addrExist = true
		case strings.HasPrefix(arg, "-newuser:"):
			userNewStr = strings.Replace(arg, "-newuser:", "", 1)
			userNewExist = true
		case strings.HasPrefix(arg, "-loaduser:"):
			userLoadStr = strings.Replace(arg, "-loaduser:", "", 1)
			userLoadExist = true
		}
```

```
	}
	if !(userNewExist || userLoadExist) || !addrExist {
		panic("failed: !(userNewExist || userLoadExist) || !addrExist")
	}
	err := json.Unmarshal([]byte(readFile(addrStr)), &Addresses)
	if err != nil {
		panic("failed: load addresses")
	}
	if len(Addresses) == 0 {
		panic("failed: len(Addresses) == 0")
	}
	if userNewExist {
		User = userNew(userNewStr)
	}
	if userLoadExist {
		User = userLoad(userLoadStr)
	}
	if User == nil {
		panic("failed: load user")
	}
}
```

Перебирает все аргументы программы, пытается найти в начале строку-команду («-loaduser:», «-newuser:», «-loadaddr:») и если находит, тогда берёт аргумент этой команды с указанием того, что команда была найдена. Если не была найдена команда работающая с пользователем или с адресами, тогда выведится ошибка, иначе глобальные переменные обновятся. Использует функции readFile (3.2), userNew (3.3), userLoad (3.4), а также глобальные переменные Addresses (3.5) и User (3.6).

(3.2) Функция readFile.

```
func readFile(filename string) string {
	data, err := ioutil.ReadFile(filename)
```

```
        if err != nil {
                return ""
        }
        return string(data)
}
```

Читает весь файл и заносит его содержимое в переменную строкового типа.

(3.3) Функция userNew.

```
func userNew(filename string) *bc.User {
        user := bc.NewUser()
        if user == nil {
                return nil
        }
        err := writeFile(filename, user.Purse())
        if err != nil {
                return nil
        }
        return user
}
```

Создаёт нового пользователя при помощи функции bc.NewUser. Также использует функцию writeFile (3.7) для создания файла и записи.

(3.4) Функция userLoad.

```
func userLoad(filename string) *bc.User {
        priv := readFile(filename)
        if priv == "" {
                return nil
        }
        user := bc.LoadUser(priv)
```

```
        if user == nil {
                return nil
        }
        return user
}
```

(3.5) Глобальная переменная Addresses.

(3.6) Глобальная переменная User.

```
var (
        Addresses []string
        User *bc.User
)
```

(3.7) Функция writeFile.

```
func writeFile(filename string, data string) error {
        return ioutil.WriteFile(filename, []byte(data), 0644)
}
```

После того как было сделано всё с инициализацией, следует приступать к консольному интерфейсу (3.8).

(3.8) Функция handleClient.

```
func handleClient() {
        var (
                message string
                splited []string
        )
```

```
for {
	message = inputString("> ")
	splited = strings.Split(message, " ")
	switch splited[0] {
	case "/exit":
		os.Exit(0)
	case "/user":
		if len(splited) < 2 {
			fmt.Println("failed: len(user) < 2\n")
			continue
		}
		switch splited[1] {
		case "address":
			userAddress()
		case "purse":
			userPurse()
		case "balance":
			userBalance()
		}
	case "/chain":
		if len(splited) < 2 {
			fmt.Println("failed: len(chain) < 2\n")
			continue
		}
		switch splited[1] {
		case "print":
			chainPrint()
		case "tx":
			chainTX(splited[1:])
		case "balance":
			chainBalance(splited[1:])
		}
	default:
		fmt.Println("command undefined\n")
	}
}
```

```
}
```

Функция состоит из бесконечного цикла, в котором содержится ввод пользователя, при помощи функции inputString (3.9). Строка разбивается по символа пробела, после чего идёт сравнение первого разделения со строковыми командами («/exit», «/user», «/chain»). Если команда не была обнаружена, выведится предупреждение. У каждой из команд, за исключением «/exit», существуют свои аргументы. Так например, у команды «/user» существуют аргументы «address» (3.10), «purse» (3.11), «balance» (3.12), а у команды «/chain» аргументы «print» (3.13), «tx» (3.14), «balance» (3.15). И каждый из этих аргументов представлен своей функцией.

(3.9) Функция inputString.

```
func inputString(begin string) string {
	fmt.Print(begin)
	msg, _ := bufio.NewReader(os.Stdin).ReadString('\n')
	return strings.Replace(msg, "\n", "", 1)
}
```

Сначала выводит сообщение, которое указано в аргументе, как строка приветствия для ввода. Далее читается строка из стандартного потока ввода до обнаружения символа новой строки. В конце, из переменной msg удаляется символ новой строки и возвращается полученное значение.

(3.10) Функция userAddress.

```
func userAddress() {
	fmt.Println("Address:", User.Address(), "\n")
}
```

(3.11) Функция userPurse.

```
func userPurse() {
	fmt.Println("Purse:", User.Purse(), "\n")
}
```

(3.12) Функция userBalance.

```
func userBalance() {
	printBalance(User.Address())
}
```

Использует функцию printBalance (3.16).

(3.13) Функция chainPrint.

```
func chainPrint() {
	for i := 0; ; i++ {
		res := nt.Send(Addresses[0], &nt.Package{
			Option: GET_BLOCK,
			Data:   fmt.Sprintf("%d", i),
		})
		if res == nil || res.Data == ""{
			break
		}
		fmt.Printf("[%d] => %s\n", i+1, res.Data)
	}
	fmt.Println()
}
```

Запрашивает у первого узла (в списке адресов) блоки, до тех пор, пока в качестве ответа не будет обнаружена пустая строка. Используется константа GET_BLOCK (3.17).

(3.14) Функция chainTX.

```
func chainTX(splited []string) {
	if len(splited) != 3 {
		fmt.Println("failed: len(splited) != 3\n")
		return
	}
	num, err := strconv.Atoi(splited[2])
	if err != nil {
		fmt.Println("failed: strconv.Atoi(num)\n")
		return
	}
	for _, addr := range Addresses {
		res := nt.Send(addr, &nt.Package{
			Option: GET_LHASH,
		})
		if res == nil {
			continue
		}
		tx := bc.NewTransaction(User, bc.Base64Decode(res.Data), splited[1],
uint64(num))
		res = nt.Send(addr, &nt.Package{
			Option: ADD_TRNSX,
			Data:   bc.SerializeTX(tx),
		})
		if res == nil {
			continue
		}
		if res.Data == "ok" {
			fmt.Printf("ok: (%s)\n", addr)
		} else {
			fmt.Printf("fail: (%s)\n", addr)
		}
	}
	fmt.Println()
```

```
}
```

В качестве аргумента «tx» предполагается два аргумента - получатель и отправляемые средства. Функция перебирает список адресов, отправляя при этом сгенерированную транзакцию под их блокчейн (подстраиваясь под последний хеш). Используются константы GET_LHASH (3.18) и ADD_TRNSX (3.19).

(3.15) Функция chainBalance.

```
func chainBalance(splited []string) {
	if len(splited) != 2 {
		fmt.Println("fail: len(splited) != 2\n")
		return
	}
	printBalance(splited[1])
}
```

(3.16) Функция printBalance.

```
func printBalance(useraddr string) {
	for _, addr := range Addresses {
		res := nt.Send(addr, &nt.Package{
			Option: GET_BLNCE,
			Data:   useraddr,
		})
		fmt.Printf("Balance (%s): %s coins\n", addr, res.Data)
	}
	fmt.Println()
}
```

Перебирает список адресов, запрашивая у каждого текущий баланс, относительно их блокчейна. Используется константа GET_BLNCE (3.20).

(3.17) Константа GET_BLOCK.

(3.18) Константа GET_LHASH.

(3.19) Константа ADD_TRNSX.

(3.20) Константа GET_BLNCE.

```
const (
        ADD_BLOCK = iota + 1
        ADD_TRNSX
        GET_BLOCK
        GET_LHASH
        GET_BLNCE
)
```

Константа ADD_BLOCK необходима для реализации узла.

(3.21) Функция main.

```
func main() {
        handleClient()
}
```

Точка входа в программу.

Данное приложение использует следующие пакеты:

```
import (
        bc "./blockchain"
        nt "./network"
        "bufio"
        "encoding/json"
        "io/ioutil"
        "fmt"
```

```
        "os"
        "strconv"
        "strings"
    )
```

3.4. Узел

Строение приложения узла очень схоже с клиентским приложением, даже некоторые функции полностью одинаковы (в исходном коде такие функции вынесены в файл «values.go»). Приложение узла имеет лишь часть с инициализацией. Таким образом, сам узел способен лишь обрабатывать информацию посылаемую клиентом, либо же другим узлом (при получении блока) на автоматизированном уровне, без ручного вмешательства.

Для запуска приложения, следует использовать следующий пример:

./node -serve::8080 -loaduser:private.key -loadchain:chain.db -loadaddr:addr.json

Точно также, как для loaduser можно применить newuser (если пользователь ещё не создан), для loadchain можно применить newchain.

Так как узел является исполняемым приложением, его пакет именуется как «main».

```
package main
```

(4.1) Функция init.

```
func init() {
	if len(os.Args) < 2 {
		panic("failed: len(os.Args) < 2")
	}
	var (
		serveStr    = ""
		addrStr     = ""
		userNewStr  = ""
		userLoadStr = ""
```

```
        chainNewStr  = ""
        chainLoadStr = ""
)
var (
        serveExist     = false
        addrExist      = false
        userNewExist   = false
        userLoadExist  = false
        chainNewExist  = false
        chainLoadExist = false
)
for i := 1; i < len(os.Args); i++ {
        arg := os.Args[i]
        switch {
        case strings.HasPrefix(arg, "-serve:"):
                serveStr = strings.Replace(arg, "-serve:", "", 1)
                serveExist = true
        case strings.HasPrefix(arg, "-loadaddr:"):
                addrStr = strings.Replace(arg, "-loadaddr:", "", 1)
                addrExist = true
        case strings.HasPrefix(arg, "-newuser:"):
                userNewStr = strings.Replace(arg, "-newuser:", "", 1)
                userNewExist = true
        case strings.HasPrefix(arg, "-loaduser:"):
                userLoadStr = strings.Replace(arg, "-loaduser:", "", 1)
                userLoadExist = true
        case strings.HasPrefix(arg, "-newchain:"):
                chainNewStr = strings.Replace(arg, "-newchain:", "", 1)
                chainNewExist = true
        case strings.HasPrefix(arg, "-loadchain:"):
                chainLoadStr = strings.Replace(arg, "-loadchain:", "", 1)
                chainLoadExist = true
        }
}
if      !(userNewExist || userLoadExist) || !(chainNewExist || chainLoadExist) ||
        !serveExist || !addrExist {
```

```
                panic("failed: !(userNewExist || userLoadExist)"+
                        "|| !(chainNewExist || chainLoadExist) || !serveExist ||
!addrExist")
        }

        Serve = serveStr
        var addresses []string
        err := json.Unmarshal([]byte(readFile(addrStr)), &addresses)
        if err != nil {
                panic("failed: load addresses")
        }
        var mapaddr = make(map[string]bool)
        for _, addr := range addresses {
                if addr == Serve {
                        continue
                }
                if _, ok := mapaddr[addr]; ok {
                        continue
                }
                mapaddr[addr] = true
                Addresses = append(Addresses, addr)
        }
        if userNewExist {
                User = userNew(userNewStr)
        }
        if userLoadExist {
                User = userLoad(userLoadStr)
        }
        if User == nil {
                panic("failed: load user")
        }
        if chainNewExist {
                Filename = chainNewStr
                Chain = chainNew(chainNewStr)
        }
        if chainLoadExist {
```

```
            Filename = chainLoadStr
            Chain = chainLoad(chainLoadStr)
        }
        if Chain == nil {
            panic("failed: load chain")
        }
        Block = bc.NewBlock(User.Address(), Chain.LastHash())
}
```

Данная функция является более модифицированной версией, по сравнению с клиентской. Здесь помимо указания адресов и приватного ключа, необходимо указать файл локальной БД и адрес с портом на принятие данных. Также используются глобальные переменные Addresses (3.5), User (3.6) и функции readFile (3.2), userNew (3.3), userLoad (3.4), как в клиентском приложении. Помимо их, используются ещё такие переменные как Chain (4.2), Block (4.3), Filename (4.4), Serve (4.5) и функции chainNew (4.6), chainLoad (4.7).

(4.2) Глобальная переменная Chain.

(4.3) Глобальная переменная Block.

(4.4) Глобальная переменная Filename.

(4.5) Глобальная переменная Serve.

```
var (
    Filename    string
    Serve       string
    Chain       *bc.BlockChain
    Block       *bc.Block
)
```

(4.6) Функция chainNew.

```
func chainNew(filename string) *bc.BlockChain {
```

```
	err := bc.NewChain(filename, User.Address())
	if err != nil {
		return nil
	}
	return bc.LoadChain(filename)
}
```

(4.7) Функция chainLoad.

```
func chainLoad(filename string) *bc.BlockChain {
	chain := bc.LoadChain(filename)
	if chain == nil {
		return nil
	}
	return chain
}
```

(4.8) Функция main.

```
func main() {
	nt.Listen(Serve, handleServer)
	for {
		fmt.Scanln()
	}
}
```

Точка входа. Использует функцию handleServer (4.9).

(4.9) Функция handleServer.

```
func handleServer(conn nt.Conn, pack *nt.Package) {
	nt.Handle(ADD_BLOCK, conn, pack, addBlock)
```

```
        nt.Handle(ADD_TRNSX, conn, pack, addTransaction)
        nt.Handle(GET_BLOCK, conn, pack, getBlock)
        nt.Handle(GET_LHASH, conn, pack, getLastHash)
        nt.Handle(GET_BLNCE, conn, pack, getBalance)
    }
```

Использует константы указанные в клиентской части (3.17), а также функции addBlock (4.10), addTransaction (4.11), getBlock (4.12), getLastHash (4.13), getBalance (4.14), .

(4.10) Функция addBlock.

```
func addBlock(pack *nt.Package) string {
        splited := strings.Split(pack.Data, SEPARATOR)
        if len(splited) != 3 {
                return "fail"
        }
        block := bc.DeserializeBlock(splited[2])
        if !block.IsValid(Chain, Chain.Size()) {
                currSize := Chain.Size()
                num, err := strconv.Atoi(splited[1])
                if err != nil {
                        return "fail"
                }
                if currSize < uint64(num) {
                        go compareChains(splited[0], uint64(num))
                        return "ok "
                }
                return "fail"
        }
        Mutex.Lock()
        Chain.AddBlock(block)
        Block = bc.NewBlock(User.Address(), Chain.LastHash())
        Mutex.Unlock()
        if IsMining {
```

```
            BreakMining <- true
            IsMining = false
        }
        return "ok"
    }
```

Принимает в данных пакета строку формата: address+SEPARATOR+chainSize+SEPARATOR+block, где address - это адрес отправителя (узла), chainSize - размер блокчейна отправителя, block - переданный блок для добавления, SEPARATOR (4.15) - константа. Возвращает строку «fail», в качестве ошибки, и строку «ok», в качестве успешного выполнения. Далее функция проверяет, является ли блок валидным по сравнению с существующим блокчейном. Если да, тогда блок добавляется в блокчейн, блок обнуляется (очищается от транзакций) и если в это время происходит майнинг (4.16), тогда майнинг прекращается и устанавливается состояние «без майнинга» (4.17). Если нет, тогда вычисляется текущий размер блокчейна. Просматривается размер блокчейна отправителя-узла и если цепь отправителя больше, тогда проверить его данные на корректность при помощи функции compareChains (4.18). Используется глобальная переменная Mutex (4.19)

(4.11) Функция addTransaction.

```
func addTransaction(pack *nt.Package) string {
    var tx = bc.DeserializeTX(pack.Data)
    if tx == nil || len(Block.Transactions) == bc.TXS_LIMIT {
        return "fail"
    }
    Mutex.Lock()
    err := Block.AddTransaction(Chain, tx)
    Mutex.Unlock()
    if err != nil {
        return "fail"
```

```
        }
        if len(Block.Transactions) == bc.TXS_LIMIT {
                go func() {
                        Mutex.Lock()
                        block := *Block
                        IsMining = true
                        Mutex.Unlock()
                        res := (&block).Accept(Chain, User, BreakMining)
                        Mutex.Lock()
                        IsMining = false
                        if res == nil && bytes.Equal(block.PrevHash, Block.PrevHash) {
                                Chain.AddBlock(&block)
                                pushBlockToNet(&block)
                        }
                        Block = bc.NewBlock(User.Address(), Chain.LastHash())
                        Mutex.Unlock()
                }()
        }
        return "ok"
}
```

Сначала идёт проверка транзакции, является ли её структура валидной и не превышен ли предел в блоке. Далее, если это последняя транзакция (после её добавления количество транзакций в блоке равняется пределу), тогда запустить параллельную функцию. В ней скопировать данные из глобальной переменной Block в локальную переменную block. Указать, что происходит майнинг, далее производить майнинг. После завершения майнинга, изменить переменную IsMining на false. Проверить результат майнинга, вернул ли метод Accept нулевой адрес и равны ли до сих пор хеши предыдущего блока в переменных block и Block (нужно при случаях, когда блок замайнился у другой ноды быстрее). Если всё успешно, тогда добавить блок в свой блокчейн и оповестить все другие узлы, о том, что создался новый блок. После всех действий - обнулить блок. Данный код использует функцию pushBlockToNet (4.20) для оповещения и пересылки блока других узлам.

(4.12) Функция getBlock.

```
func getBlock(pack *nt.Package) string {
	num, err := strconv.Atoi(pack.Data)
	if err != nil {
		return ""
	}
	size := Chain.Size()
	if uint64(num) < size {
		return selectBlock(Chain, num)
	}
	return ""
}
```

Использует функцию selectBlock (4.21).

(4.13) Функция getLastHash.

```
func getLastHash(pack *nt.Package) string {
	return bc.Base64Encode(Chain.LastHash())
}
```

(4.14) Функция getBalance.

```
func getBalance(pack *nt.Package) string {
	return fmt.Sprintf("%d", Chain.Balance(pack.Data, Chain.Size()))
}
```

(4.15) Константа SEPARATOR.

```
const (
	SEPARATOR = "_SEPARATOR_"
)
```

(4.16) Глобальная переменная IsMining.

(4.17) Глобальная переменная BreakMining.

```
var (
	IsMining bool
	BreakMining = make(chan bool)
)
```

(4.18) Функция compareChains.

```
func compareChains(address string, num uint64) {
	filename := "temp_" + hex.EncodeToString(bc.GenerateRandomBytes(8))
	file, err := os.Create(filename)
	if err != nil {
		return
	}
	file.Close()
	defer func() {
		os.Remove(filename)
	}()
	res := nt.Send(address, &nt.Package{
		Option: GET_BLOCK,
		Data:   fmt.Sprintf("%d", 0),
	})
	if res == nil {
		return
	}
	genesis := bc.DeserializeBlock(res.Data)
	if genesis == nil {
```

```
            return
        }
        db, err := sql.Open("sqlite3", filename)
        if err != nil {
            return
        }
        defer db.Close()
        _, err = db.Exec(bc.CREATE_TABLE)
        chain := &bc.BlockChain{
            DB: db,
        }
        chain.AddBlock(genesis)
        for i := uint64(1); i < num; i++ {
            res := nt.Send(address, &nt.Package{
                Option: GET_BLOCK,
                Data:   fmt.Sprintf("%d", i),
            })
            if res == nil {
                return
            }
            block := bc.DeserializeBlock(res.Data)
            if block == nil {
                return
            }
            if !block.IsValid(chain, i) {
                return
            }
            chain.AddBlock(block)
        }
        Mutex.Lock()
        Chain.DB.Close()
        os.Remove(Filename)
        copyFile(filename, Filename)
        Chain = bc.LoadChain(Filename)
        Block = bc.NewBlock(User.Address(), Chain.LastHash())
        Mutex.Unlock()
```

```
        if IsMining {
                BreakMining <- true
                IsMining = false
        }
}
```

Создаёт временный файл, запрашивает у узла генезис-блок, вносит генезис-блок в файл, как в локальную БД. Далее, через цикл, запрашивает блоки до указанного максимума (размера блокчейна ноды). Если все блоки прошли проверку, тогда удалить текущий файл связанный с блокчейном и заменить его на временный файл. Далее загрузить новый блокчейн и обнулить блок. Проверить, запущен ли майнинг, и если да, тогда остановить его. Использует функцию copyFile (4.22).

(4.19) Глобальная переменная Mutex.

```
var (
        Mutex sync.Mutex
)
```

Необходима для блокировки параллельных частей кода, которые изменяют состояние программы методом редактирования глобальных переменных.

(4.20) Функция pushBlockToNet.

```
func pushBlockToNet(block *bc.Block) {
        var (
                sblock = bc.SerializeBlock(block)
                msg = Serve + SEPARATOR + fmt.Sprintf("%d", Chain.Size()) +
        SEPARATOR + sblock
        )
        for _, addr := range Addresses {
```

```
                go nt.Send(addr, &nt.Package{
                        Option: ADD_BLOCK,
                        Data:   msg,
                })
        }
}
```

(4.21) Функция selectBlock.

```
func selectBlock(chain *bc.BlockChain, i int) string {
        var block string
        row := chain.DB.QueryRow("SELECT Block FROM BlockChain WHERE Id=$1",
i+1)
        row.Scan(&block)
        return block
}
```

(4.22) Функция copyFile.

```
func copyFile(src, dst string) error {
        in, err := os.Open(src)
        if err != nil {
                return err
        }
        defer in.Close()
        out, err := os.Create(dst)
        if err != nil {
                return err
        }
        defer out.Close()
        _, err = io.Copy(out, in)
        if err != nil {
                return err
```

```
        }
        return out.Close()
    }
```

Данное приложение использует следующие пакеты:

```
    import (
        bc "./blockchain"
        nt "./network"
        "encoding/json"
        "fmt"
        "os"
        "strings"
        "io/ioutil"
        "bytes"
        "database/sql"
        "encoding/hex"
        _ "github.com/mattn/go-sqlite3"
        "io"
        "strconv"
        "sync"
    )
```

Компиляция клиента и узла:

```
    go build client.go
    go build node.go
```

Теперь, чтобы проверить работоспособность клиента и узла, можно запустить три процесса, два из которых будут представлять майнеров (или же узлов) и один будет являться клиентом. Нумерация процессов будет начинаться

с единицы и номер будет указываться в фигурных скобках (например, третий процесс = {3}).

Предполагается, что у всех пользователей уже имеется файл с адресами узлов (addr.json). Запуск приложений выглядит следующим образом:

```
{1} ./node -serve::8080 -newuser:node1.key -newchain:chain1.db -loadaddr:addr.json
{2} ./node -serve::9090 -newuser:node2.key -newchain:chain2.db -loadaddr:addr.json
{3} ./client -loaduser:node1.key -loadaddr:addr.json
```

Из этих запусков видно, что используется дважды один и тот же файл с приватным ключом (node1.key). Иными словами, пользователь, обладающий данным приватным ключом, будет являться одновременно узлом, способным майнить блоки, и клиентом, который будет создавать транзакции. Из этого примера будет вытекать сразу же противоречие между двумя майнерами, так как в одном блокчейне пользователь имеет баланс (так как он же является создателем генезис-блока), а в другом его баланс равен нулю. Соответственно, после майнинга нового блока (первым узлом) должен будет произойти soft fork, который отклонит блокчейн у одного майнера и примет блокчейн второго майнера. После этого, действия майнеров синхронизируются и соответственно появится возможность одновременного майнинга на разных узлах.

Запустив все три процесса, только процесс связанный с клиентом {3} отобразит ожидание ввода из терминала. Написав команду «/user balance» можно просмотреть баланс текущего пользователя относительно блокчейна у разных узлов:

```
> /user balance
Balance (:8080): 100 coins
Balance (:9090): 0 coins
```

В данном примере существует противоречие двух разных блокчейнов, генезис-блоки которых имеют разных майнеров. Чтобы решить данное противоречие, необходимо сделать один блокчейн больше другого.

Соответственно нужно первому узлу отправить транзакции, для их добавления в блок и последующего его майнинга. Это можно осуществить командой «/chain tx <receiver> <value>».

> /chain tx aaa 3
ok: (:8080)
fail: (:9090)

> /chain tx bbb 2
ok: (:8080)
fail: (:9090)

Первый узел будет успешно принимать транзакции, в то время как второй, их будет отвергать (из-за того, что по его блокчейну у пользователя недостаточно средств для отправки средств).

Если посмотреть на процесс первого узла {1}, в момент занесения последней транзакции, то можно увидеть, что он майнит блок. Результат майнинга:

Mining: AAANWmdWf/TVO8BgO82ItoFObf1z9pdorinr4dd7yvg=

После майнинга, блокчейн двух узлов должен синхронизироваться. Успешность «срастания» блокчейнов можно наблюдать со стороны клиента, если он снова захочет узнать текущий свой баланс.

> /user balance
Balance (:8080): 96 coins
Balance (:9090): 96 coins

Пользователь имел баланс равный 100 монетам, до создания блока. Потратил он 5 монет (3 отправились пользователю «ааа», 2 отправились

пользователю «bbb»). Но так как он же и является майнером, ему из хранилища передалась 1 монета за успешный майнинг. Таким же образом можно посмотреть баланс самого хранилища, при помощи команды «/chain balance STORAGE-CHAIN».

> /chain balance STORAGE-CHAIN
Balance (:8080): 99 coins
Balance (:9090): 99 coins

После успешной синхронизации узлов появится неизбежно и конкуренция за майнинг блока. Создадим ещё две транзакции для генерации блока.

> /chain tx qqq 1
ok: (:8080)
ok: (:9090)

> /chain tx www 1
ok: (:8080)
ok: (:9090)

На этот раз, все два узла приняли транзакции и начали одновременный майнинг. Это можно наблюдать в процессах {1} и {2}.

{1} Mining: WfJq7U4MI/ZyCAfFrLEIoX84HmoJaO9PnOkzI+8Hchw=
{2} Mining: AAABb7uK/R3/rDcCDel4o7S6m1vFDYS5UsJBIOAV0mU=

Ситуация «выигрыша» неопределена и носит вероятностный характер, так как один узел может начать с более лучшей позиции значения nonce или обладать б*о*льшими вычислительными мощностями, чем другой узел. В данном случае, в гонке за майнинг блока победил второй процесс {2}, при этом первый

процесс {1} сразу же прекратил вычисление хеша, как только принял информацию о найденном хеше другим узлом.

Если посмотреть текущий баланс пользователя, то он будет равен 94 монетам, то-есть награду за майнинг данный пользователь не получил. При этом, если посмотреть баланс майнера, который вычислил блок, то он будет равен одной монете.

Помимо просмотра баланса (как своего, так и других пользователей) и создания транзакций, можно посмотреть весь блокчейн при помощи команды «/chain print».

```
> /chain print
...
[3] => {
        "Nonce": 3177602599,
        "Difficulty": 20,
        "CurrHash": "qcYXjw4xnhFt7KuDj5VFUYAevJ8cf0N0LtxDkYCDZ/k=",
        "PrevHash": "uj3RWYXqgN7+4Hh0kGJa8MmU5BQ2AmcEBVOsePzrOGU=",
        "Transactions": [
                {
                        "RandBytes":
"m0dXqyVNL6lHztZTaZM4zgnMCzDPazNRo+UtGtGrhh0=",
                        "PrevBlock":
"uj3RWYXqgN7+4Hh0kGJa8MmU5BQ2AmcEBVOsePzrOGU=",
                        "Sender":
"MEgCQQDY38h314lRok0tncoyk9tc/5UG0quwsuZZt50WzI/Xa6Nx0bErz5TX0+VMTx2l8FLW5
V5HRdEwSLCrZkbMeMIbAgMBAAE=",
                        "Receiver": "qqq",
                        "Value": 1,
                        "ToStorage": 0,
                        "CurrHash":
"2n9MXXM2j/awo/+aZHmx2vGz6o87VGicFSvkyl07JCg=",
                        "Signature":
"D6roOQ5/6lp8cW+xxOjS0Xip+9AzPeNOi/Aorml7sFmBtyuHyxWtD/25EeU2RKEpn2w1PduUP
y58iCosTA8ZAw=="
```

```
                },
                {
                        "RandBytes":
"3C72TAcATdyAcLzMYyQvzHnrXnK9lUC2zvGaGH7/BbA=",
                        "PrevBlock":
"uj3RWYXqgN7+4Hh0kGJa8MmU5BQ2AmcEBVOsePzrOGU=",
                        "Sender":
"MEgCQQDY38h314lRok0tncoyk9tc/5UG0quwsuZZt50WzI/Xa6Nx0bErz5TX0+VMTx2l8FLW5
V5HRdEwSLCrZkbMeMIbAgMBAAE=",
                        "Receiver": "www",
                        "Value": 1,
                        "ToStorage": 0,
                        "CurrHash":
"OUv6zdMAIXXUM5uCRvLSkL9Zw1stOk3BoUkDmSsqeG0=",
                        "Signature":
"nFKsei0rWYsRzoVcnS7GrOK76Jn48IFt56xNpkOOReYnmMGh2Vbz2kvkONe703z2arVi98chF
Ex2IdytBXlLhw=="
                },
                {
                        "RandBytes":
"M5ptIBYxjmd2vtY0eJORhqJVJFoeMZrewmS0c+VEHAI=",
                        "PrevBlock": null,
                        "Sender": "STORAGE-CHAIN",
                        "Receiver":
"MEgCQQDU6JaPgXMlugPTqNrcOqHj9WF5WEYC8IyEtSY8onEFIHeLiK8U3lrKZ9la3K6tuPD
mrat9s8qXR50gnZ8ngcOZAgMBAAE=",
                        "Value": 1,
                        "ToStorage": 0,
                        "CurrHash": null,
                        "Signature": null
                }
        ],
        "Mapping": {
    "MEgCQQDU6JaPgXMlugPTqNrcOqHj9WF5WEYC8IyEtSY8onEFIHeLiK8U3lrKZ9la3
K6tuPDmrat9s8qXR50gnZ8ngcOZAgMBAAE=": 1,
```

"MEgCQQDY38h314lRok0tncoyk9tc/5UG0quwsuZZt50WzI/Xa6Nx0bErz5TX0+VMTx2l8FLW5V5HRdEwSLCrZkbMeMIbAgMBAAE=": 94,
"STORAGE-CHAIN": 98,
"qqq": 1,
"www": 1
},
"Miner": "MEgCQQDU6JaPgXMlugPTqNrcOqHj9WF5WEYC8IyEtSY8onEFIHeLiK8U3lrKZ9la3K6tuPDmrat9s8qXR50gnZ8ngcOZAgMBAAE=",
"Signature": "V0aX0zwR4CX+OmRWKPX5g4fPZ2hqDm0Kg204UUmiaXoE4PFEasl5EudRGP8MdQsjRcTTxwQQuFUc681plf/s/w==",
"TimeStamp": "2020-07-17T18:17:20-04:00"
}

Вывод сократил до одного блока. В его состоянии можно как раз наблюдать балансы пользователей, а в транзакциях награду за майнинг.

Чтобы выйти из клиентского приложения, можно воспользоваться командой «/exit».

```
> /exit
```

---

4. Литература

[1] Абельсон, Х., Сассман, Дж. Структура и интерпретация компьютерных программ / Х. Абельсон, Дж. Сассман. - М.: Добросвет, КДУ, 2018. - 608 с.

[2] Шнайер, Б. Прикладная криптография. Протоколы, алгоритмы и исходные коды на языке С / Б. Шнайер. - СпБ.: ООО «Альфа-книга», 2018. - 1040 с.

[3] Шнайер, Б., Фергюсон, Н. Т. Практическая криптография / Б. Шнайер, Н. Т. Фергюсон. - М.: Издательский дом «Вильямс, 2005. - 420 с.

[4] Рябко, Б. Я., Фионов А. Н. Криптография в информационном мире / Б. Я. Рябко, А.Н. Фионов - М.: Горячая линия - Телеком, 2019. - 300 с.

[5] Керниган, Б. У., Донован, Ф. Ф. Язык программировани Go / Б. У. Керниган, А. А. Донован. - М.: ООО «И.Д. Вильямс», 2018. - 432 с.

5. Исходный код

https://github.com/Number571/Blockchain