

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	9
1. РЕШЕНИЯ ДЛЯ ДЕЦЕНТРАЛИЗОВАННОГО ОБМЕНА ДАНЫМИ И ТЕХНОЛОГИЯ BLOCKCHAIN	11
1.1. Обзор существующих решений для децентрализованного обмена данными	11
1.2. Технология blockchain	12
1.3. Обзор существующих blockchain решений	13
2. ПОСТРОЕНИЕ СТРУКТУРЫ РЕЕСТРА И ВЫБОР ТЕХНОЛОГИЙ.....	15
2.1. Выбор хеш-функции	15
2.2. Выбор средств криптографии.....	17
2.3. Структура сети	20
2.4. Транзакции.....	24
2.5. Алгоритм консенсуса.....	25
2.6. Генезис-блок.....	29
2.7. Структура блоков.....	30
2.8. Добавление новых данных в реестр.....	31
3. РАЗРАБОТКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ДЛЯ ОРГАНИЗАЦИИ РЕЕСТРА.....	33
3.1. Выбор средств разработки	33
3.2. Разрабатываемые приложения.....	34
3.3. Структура консольного приложения	36
3.4. Формат данных.....	37

3.5. Реализация сетевого взаимодействия	41
3.5.1. Выбор протокола передачи данных.....	41
3.5.2. Проектирование API.....	42
3.5.3. Получение списка мастер-узлов	48
3.6. Реализация взаимодействия с цепочкой блоков	51
3.7. Создание новой сети.....	52
3.7.1. Создание и хранение ключей	52
3.7.2. Создание генезис-блока	54
3.8. Установка и запуск консольного приложения.....	55
3.9. Создание приложения с графическим интерфейсом.....	56
3.9.1. Проектирование и разработка	56
3.9.2. Графический интерфейс	58
3.9.3. Установка	66
3.10. Разработка веб-приложения	67
ЗАКЛЮЧЕНИЕ	71
СПИСОК ЛИТЕРАТУРЫ	73

ВВЕДЕНИЕ

В современном мире информационные технологии стремительно развиваются и становятся более доступными. Благодаря этому они всё чаще используются как государствами, так и обычными людьми. В связи с этим многие государства начали предоставлять гражданам доступ к открытым электронным реестрам. Например, в Российской Федерации через интернет можно получить доступ к реестру программ для ЭВМ, реестру товарных знаков и знаков обслуживания, единому государственному реестру юридических лиц и так далее.

Безусловно, электронные реестры имеют ряд преимуществ:

1. Экономия времени. Вместо поездки в офис приёма-выдачи документов и ожидания в очереди достаточно заполнить заявку на сайте или найти интересующую информацию самостоятельно.

2. Экономия средств. Зачастую выписка из реестра в электронном виде или полностью бесплатна, или дешевле бумажного варианта.

Несмотря на все плюсы, у электронных реестров имеются и свои недостатки:

Большое количество современных публичных реестров как правило построены по централизованному принципу с резервированием и одновременно работа происходит только с одним из серверов. Поэтому, несмотря на все плюсы, у таких электронных реестров имеются и свои недостатки.

1. Ненадёжность. Если основной и резервный сервер выйдут из строя, реестр станет полностью недоступен. Если же резервирования нет, доступ ко всей информации будет потерян при неполадках даже на одном сервере.

2. Непрозрачность. В таком реестре информация в любой момент может быть изменена без чьего-либо ведома. При этом доступ к старым данным будет навсегда утерян.

Соответственно, для обеспечения большей надёжности и безопасности информации следует организовывать публичные реестры по децентрализованному принципу.

Главной целью данной работы является разработка программного обеспечения для организации публичного децентрализованного реестра с применением технологии blockchain.

В работе рассмотрены существующие решения для распределённого хранения информации и причины, по которым они не могут быть применены для создания открытых реестров. Приведено описание технологии blockchain и обоснование её использования. Предложен подход к реализации и вариант построения системы распределенного хранения данных. Произведён сравнительный анализ и выбор технологий для реализации ПО. Рассмотрены алгоритмы взаимодействия элементов системы и криптографические методы для обеспечения безопасного хранения и передачи данных. Разработано кроссплатформенное программное обеспечение для организации публичного децентрализованного реестра и графический интерфейс для удобной работы с ним.

1. РЕШЕНИЯ ДЛЯ ДЕЦЕНТРАЛИЗОВАННОГО ОБМЕНА ДАНЫМИ И ТЕХНОЛОГИЯ BLOCKCHAIN

1.1. Обзор существующих решений для децентрализованного обмена данными

В настоящее время уже существуют решения для децентрализованного обмена данными. Наиболее используемые из них: Gnutella и BitTorrent [1]. Обе технологии являются протоколами для распределённого обмена файлами между пользователями и позволяют загружать файлы с компьютеров других участников сети без использования центрального сервера.

Также стоит внимания технология IPFS (InterPlanetary File System). IPFS – это протокол, объединяющий всех участников сети в единую файловую систему и позволяющий удобно получать доступ к файлам с помощью собственного пространства имён [2].

Данные протоколы отлично справляются со своей задачей, однако, по ряду причин они не подходят для создания реестров.

Во-первых, в большинстве случаев для реестров необходимо обеспечить достоверность последовательности добавляемой в них информации. Например, для реестра недвижимости нужно обеспечить правильный порядок осуществления сделок, чтобы знать, какая из них произошла раньше, а какая позже. Иначе такой реестр не будет иметь смысла.

Во-вторых, в перечисленных технологиях добавлять информацию может абсолютно любой участник сети. В реестры же вносить данные могут только определённые люди. Соответственно, для организации реестра необходимо предоставить возможность добавлять записи только ограниченному числу человек.

Для решения вышеперечисленных проблем предлагается использовать технологию blockchain.

1.2. Технология blockchain

Blockchain – это цепочка блоков. В его минимальной реализации каждый блок содержит некоторую хранимую информацию, хеш-сумму этого блока и хеш-сумму предыдущего блока [3]. С помощью контрольных хеш-сумм обеспечивается правильная последовательность блоков. Если третье лицо захочет изменить информацию в одном блоке, ему необходимо будет пересчитать и заменить контрольные суммы во всех блоках, стоящих после изменённого. Однако сделать это не получится, так как система децентрализованная и остальные участники уже хранят всю достоверную цепочку блоков и проверяют соответствие хешей. На рисунке 1.1 изображён пример пересчёта хеш-сумм при изменении информации. При попытке внести в цепочку блок с недостоверным хешем, остальные узлы отклоняют этот блок.

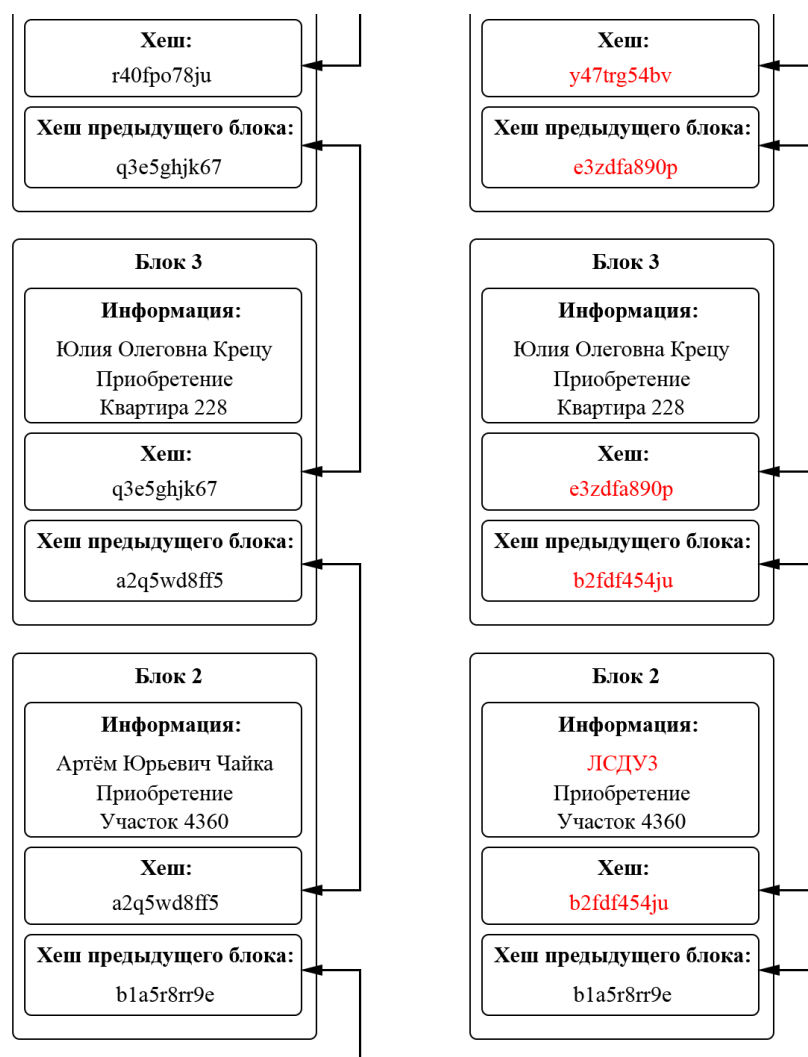


Рисунок 1.1 Пример пересчёта хеш-сумм

1.3. Обзор существующих blockchain решений

На сегодняшний день существуют blockchain решения, которые потенциально позволяют реализовать публичный децентрализованный реестр.

В первую очередь стоит упомянуть технологию STORJ – распределённое облачное хранилище на blockchain, которое позволяет арендовать дисковое пространство у других пользователей. Однако аренда является платной, и blockchain в данном случае используется для оплаты услуг и распределения файлов между участниками сети [4].

Также в blockchain платформах Ethereum, EOS, Tron возможно создание распределённых приложений на основе blockchain с помощью смарт-контрактов.

Главным преимуществом Ethereum является наибольшее количество доступных приложений [5]. Однако обработка новой информации в приложениях может занимать достаточно продолжительное время.

EOS же предоставляет наибольшую скорость работы из всех перечисленных технологий, однако количество доступных приложений существенно ниже, чем у Ethereum.

Tron обычно используется для создания игровых и развлекательных приложений.

Смарт-контракт – это алгоритм, который выполняется децентрализованно [6]. Результат работы алгоритма может проверить каждый участник сети, так как смарт-контракт и данные, которые он использует, хранятся в цепочке блоков. С помощью смарт-контрактов возможно создавать различные распределённые приложения. Например, пользуются популярностью децентрализованные обменные биржи, азартные и коллекционные игры, финансовые приложения. В перечисленных выше сетях смарт-контракты обладают полнотой по Тьюрингу. Это значит, что с их

помощью имеется возможность реализовывать любые вычислительные функции. Соответственно, такие контракты позволяют хранить абсолютно любую информацию в цепочке блоков.

Однако у такого подхода имеется один существенный недостаток – приведённые выше технологии в первую очередь используются как децентрализованные платёжные системы. Поэтому, каждый раз, когда пользователь добавляет информацию в такую сеть, ему приходится платить комиссию. Использование же реестра должно быть полностью бесплатным, следовательно полностью готовые решения на базе смарт-контрактов не подойдут для наших целей.

Так как приведённые выше решения представляют собой программное обеспечение с открытым исходным кодом, всё же имеется возможность модифицировать их для создания реестра. Однако данные решения имеют огромную кодовую базу, на разбор которой уйдёт большое количество времени, а интерфейс для удобной работы с реестром в любом случае придётся писать самому.

Учитывая все приведённые выше факторы, было принято решение разрабатывать программное обеспечение для реализации публичного децентрализованного реестра с нуля.

2. ПОСТРОЕНИЕ СТРУКТУРЫ РЕЕСТРА И ВЫБОР ТЕХНОЛОГИЙ

2.1. Выбор хеш-функции

Как упоминалось ранее, хеш-суммы являются основой технологии blockchain.

Хеш-функция – это функция, которая преобразовывает массив данных произвольной длины в строку определённой длины – хеш-сумму. Данный процесс называется хешированием.

С помощью хеш-сумм можно идентифицировать был ли изменён набор данных. Например, строка «Hello World!» с помощью хеш-функции MD5 будет преобразована в «ed076287532e86365e841e92bfc50d8c». Если же изменить в ней всего лишь один символ, мы получим совершенно другую хеш-сумму. Так, строка «Hello World» будет преобразована в «b10a8db164e0754105b7a99be72e3fe5».

На данный момент существует большое количество различных хеш-функций. Самые используемые из них: MD5 и семейство функций SHA-2, которое включает себя такие функции, как SHA-224, SHA-256, SHA-384, SHA-512 и другие.

Самый важный критерий при выборе функции – устойчивость к коллизиям. Коллизия – это получение одинаковой хеш-суммы из различных входных данных. Теоретически в любом алгоритме хеширования возможно появление коллизий, так как количество комбинаций входных данных больше, чем количество возможных хеш-сумм. Объясняется это тем, что длина входных данных может быть любой, а длина хеш-суммы фиксированная. Поэтому считается, что хеш-функция является устойчивой, если для неё не существует алгоритмов подбора коллизий за обозримое время. MD5 и SHA-1 не являются устойчивыми [7, 8], поэтому не будут рассмотрены в качестве кандидатов.

Так как в blockchain хеширование происходит довольно часто, скорость вычисления хеш-суммы является важным критерием. Для определения самой быстрой функции были написан следующий тест:

1. Начинаем измерение.
2. Генерируем случайные 1024 байта.
3. Производим хеширование этих данных.
4. Повторяем первый и второй пункт 1 000 000 раз.
5. Заканчиваем измерение.
6. Повторяем приведённые выше пункты для каждой функции.

Результаты теста приведены на рисунке 2.1.

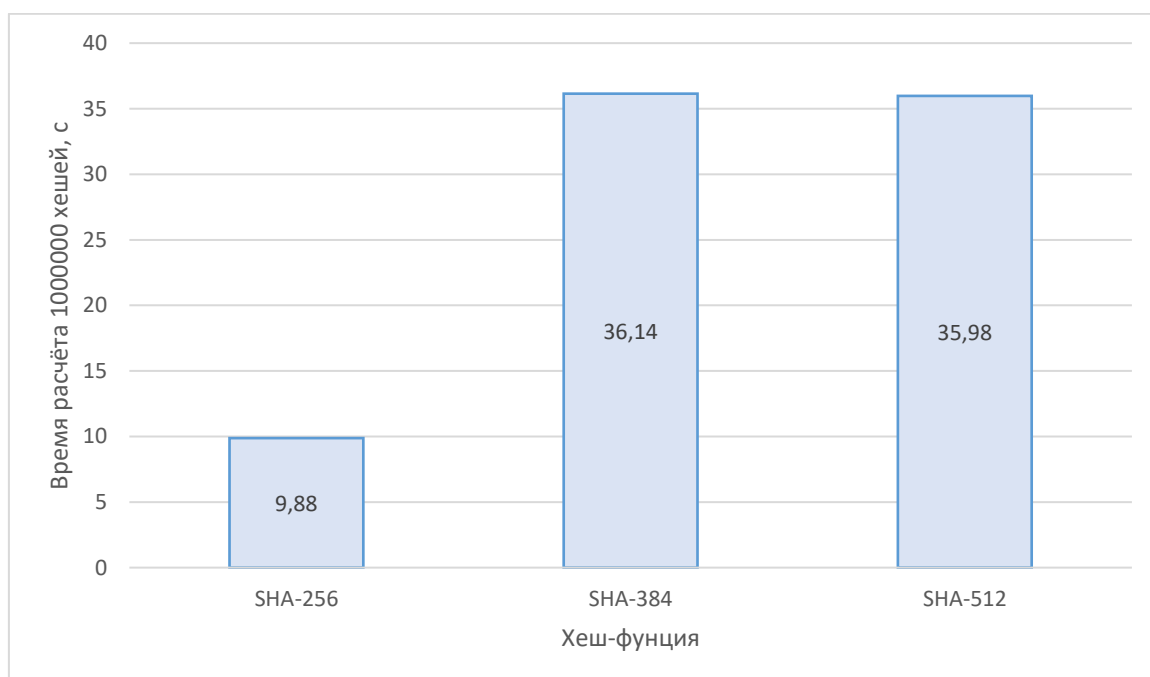


Рисунок 2.1 Скорость хеширования различными функциями

Из данного теста видно, что функция SHA-256 рассчитала один миллион хешей почти в 4 раза быстрее, чем функции SHA-384 и SHA-512. Следовательно, по параметру скорости следует выбрать именно SHA-256.

Так как в blockchain хранится и передаётся по сети большое количество хеш-сумм, немаловажным критерием выбора является её длина. На рисунке 2.2 показано сравнение размеров хеш-сумм разных функций.

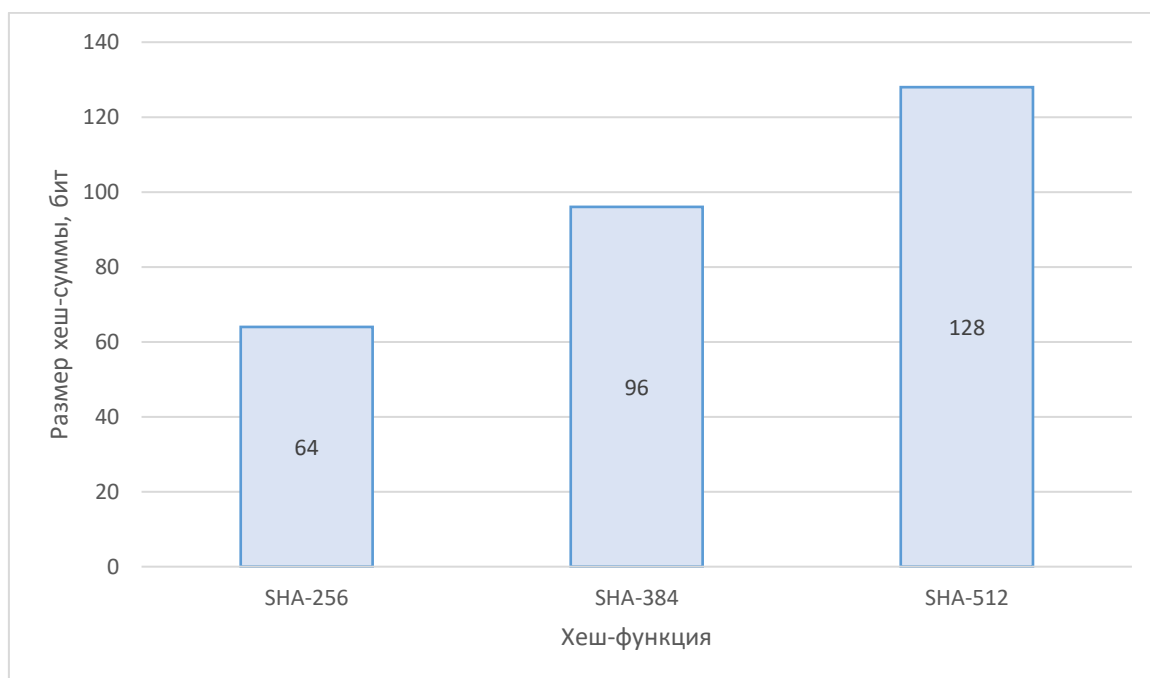


Рисунок 2.2 Размер хеш-сумм различных функций

Как мы видим, длина хеш-суммы SHA-256 составляет 64 бита, что в 1.5 раза меньше, чем у SHA-384 и в 2 раза меньше, чем у SHA-512. Соответственно, по критерию размера хеша следует выбрать функцию SHA-256.

Можно сделать вывод, что, исходя из критериев скорости хеширования, длины хеш-суммы и устойчивости к коллизиям, для создания реестра следует использовать хеш-функцию SHA-256. К тому же SHA-256 является самой используемой функцией в существующих blockchain проектах [9].

2.2. Выбор средств криптографии

Как было сказано в пункте 1.1 работы, для организации реестра необходимо предоставить возможность добавлять записи только ограниченному числу человек. Реализовать данную возможность можно с

помощью применения криптографических систем с открытым ключом. Такие системы также называются асимметричными.

В асимметричных криптосистемах существует два вида ключей: открытые (публичные) и закрытые (приватные). Такие системы могут быть использованы как для шифрования, так и для реализации цифровой подписи. Нам необходим второй вариант. Электронная подпись работает по следующему принципу:

1. Алиса генерирует пару ключей (открытый и закрытый).
2. Алиса хранит в тайне закрытый ключ и подписывает с помощью него отправляемые сообщения.
3. Алиса посылает Бобу сообщение, цифровую подпись и свой открытый ключ.
4. Боб с помощью цифровой подписи проверяет, соответствует ли цифровая подпись сообщению и открытому ключу.

Таким образом, только Алиса сможет отправить вместе с сообщением правильную электронную подпись, потому что только она имеет доступ к закрытому ключу. К тому же, если злоумышленник перехватит сообщение и захочет изменить его, цифровая подпись перестанет быть правильной. Это значит, что электронная подпись гарантирует авторство сообщения и его целостность.

Однако сгенерировать для себя валидную пару ключей может любой пользователь, поэтому нам необходимо знать и хранить открытые ключи именно тех, кому разрешено добавлять информацию в реестр. Подробнее об этом написано в пункте 2.6.

В настоящее время самыми известными асимметричными криптосистемами являются RSA, DSA, а также системы, основанные на эллиптических кривых.

Для реализации реестра предлагается использовать криптосистемы, основанные на эллиптических кривых.

Самое главное преимущество эллиптической криптографии – длина открытого ключа. Для децентрализованного реестра этот параметр очень важен, так как открытый ключ будет содержаться в каждом сообщении, а значит храниться в цепочке блоков и передаваться по сети. Соответственно, чем меньше этот параметр при равной криптостойкости, тем лучше.

В таблице 2.1 показано, какой должна быть длина открытого ключа в различных системах для обеспечения сопоставимого уровня безопасности по результатам исследований [10].

Таблица 2.1 – длина открытых ключей в битах

Эллиптическая криптография	RSA
163	1024
233	2240
283	3072
409	7680
571	15360

Как видно из таблицы, размер ключей при использовании эллиптической криптографии как минимум в 6 раз меньше, чем у RSA. А при повышении требуемого уровня защиты разницы между ними может достигать 26 раз. Таким образом, использование криптографии на эллиптических кривых поможет существенно сэкономить пропускную способность сети и используемое дисковое пространство.

Существует множество различных эллиптических кривых и схем подписи. Для реализации реестра предлагается использовать Ed25519 – схему подписи EdDSA, основанную на эллиптической кривой Curve25519 и использующую SHA-512. Данная схема имеет ряд преимуществ. Во-первых, эллиптическая кривая Curve25519 считается полностью безопасной [11]. Во-вторых, в этой схеме подписи не используются ветвления и операции с памятью, позволяющие выполнять атаки с использованием информации о

физических процессах в устройстве. В-третьих, во время подписи не используется генератор случайных чисел, что предотвращает воссоздание приватного ключа по подписи в случае предсказуемости генератора [12].

Закрытый ключ предлагается использовать длиной 128 бит. В таком случае длина открытого ключа будет составлять 256 бит. Размер же цифровой подписи будет равняться 512 битам. Как говорилось ранее, для обеспечения примерно такого же уровня безопасности при использовании RSA потребовались бы открытые ключи длиной в 3072 бита, что в 12 раз больше, чем у Ed25519.

2.3. Структура сети

В настоящее время большинство приложений, которые мы используем, построены с применением клиент-серверной архитектуры. В таких сетях для получения и отправки информации клиенты обращаются к одному центральному серверу. Пример такой сети приведён на рисунке 2.3.

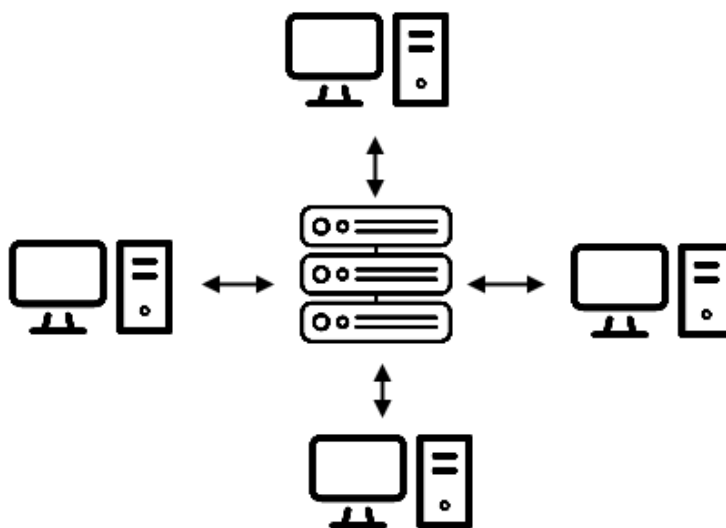


Рисунок 2.3 Пример клиент-серверной архитектуры

Недостатки такого подхода были приведены в начале работы. Для организации реестра предлагается использовать P2P (peer-to-peer) сеть. Такие сети также часто называют одноранговыми или пиринговыми. В отличие от клиент-серверной архитектуры, в таких сетях отсутствуют центральные

сервера, и все участники сети взаимодействуют напрямую между собой, что позволяет увеличить устойчивость и надёжность сети.

Существует два основных типа пиринговых сетей: структурированные и неструктурированные. В структурированных сетях используется маршрутизация, поэтому взаимодействие происходит только с определёнными узлами. В неструктурированных сетях все участники взаимодействуют друг с другом [13]. При разработке реестра было принято решение использовать неструктурированную сеть. Несмотря на увеличение объёма трафика, такой вариант позволяет увеличить устойчивость сети. Пример неструктурированной одноранговой сети приведён на рисунке 2.4.

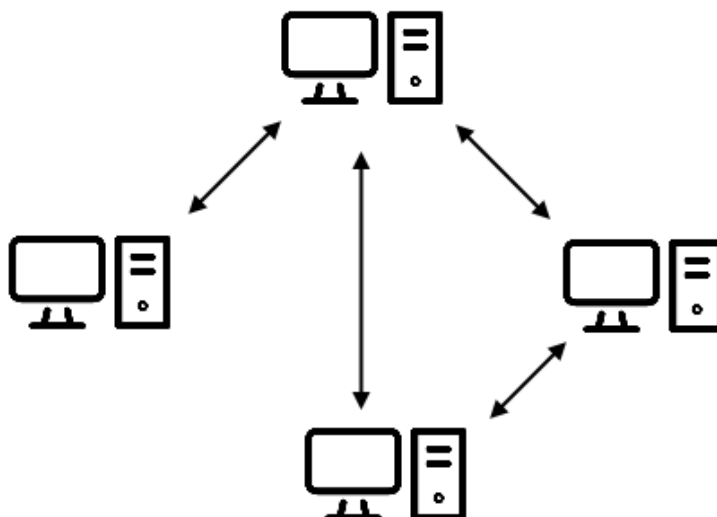


Рисунок 2.4 Пример неструктурированной peer-to-peer сети.

С сетевой точки зрения предлагается использовать два типа узлов: обычные узлы хранения и мастер-узлы. Основой сети являются мастер-узлы – они имеют прямой выход в глобальную сеть. Обычные же узлы хранения недоступны для остальных извне, так как не имеют белого IP-адреса, находятся за NAT или по какой-либо другой причине. Соответственно, для получения и отправки информации такие узлы (клиенты) обращаются к мастер-узлам, которые в данном случае являются серверами. Мастер-узлы могут выступать и в роли сервера, и в роли клиента, так как точно так же обращаются к другим мастер-узлам для получения и отправки новых данных.

Помимо новой информации, мастер-узлы также предоставляют информацию о известных им мастер-узлах. Это позволяет масштабировать и поддерживать сеть в рабочем состоянии.

Рассмотрим на примере, как происходит взаимодействие с сетевой точки зрения:

1. Все обычные узлы знают адреса нескольких мастер-узлов по умолчанию. Допустим, таких мастер-узлов два (в настоящей сети их должно быть больше).

2. Мастер-узлы запущены и успешно взаимодействуют друг с другом.

3. Клиент подключается к сети и пытается установить соединение с мастер-узлом 1 и мастер-узлом 2. Допустим, по какой-либо причине соединение между ним и мастер-узлом 2 отсутствует и ему удалось подключиться только к мастер-узлу 1.

4. К сети в первый раз подключается мастер-узел 3. Соответственно, он, как и любой клиент, пытается установить соединение с мастер-узлом 1 и мастер-узлом 2. При этом он сообщает, что является мастер узлом и передаёт им свой адрес. Допустим, по какой-либо причине соединение между ним и мастер-узлом 1 отсутствует и ему удалось подключиться только к мастер-узлу 2.

5. Теперь несмотря на то, что соединение между мастер-узлом 1 и мастер-узлом 3 отсутствует, мастер-узел 2 передаст мастер-узлу 1 информацию о мастер-узле 3. А когда клиент запросит у мастер-узла 1 новую информацию, мастер-узел 1 отправит клиенту информацию о появлении мастер-узла 3 и его адрес.

6. Теперь клиент может подключиться и к мастер-узлу 3. К тому же теперь каждый раз при следующих включениях, он будет автоматически пытаться соединиться с ним, так же как к мастер-узлу 1 и мастер-узлу 2.

На рисунке 2.5 поэтапно показан этот процесс подключения нового мастер-узла к сети. Стрелочками обозначены установленные соединения. Передача информации и неудачные попытки подключения опущены.

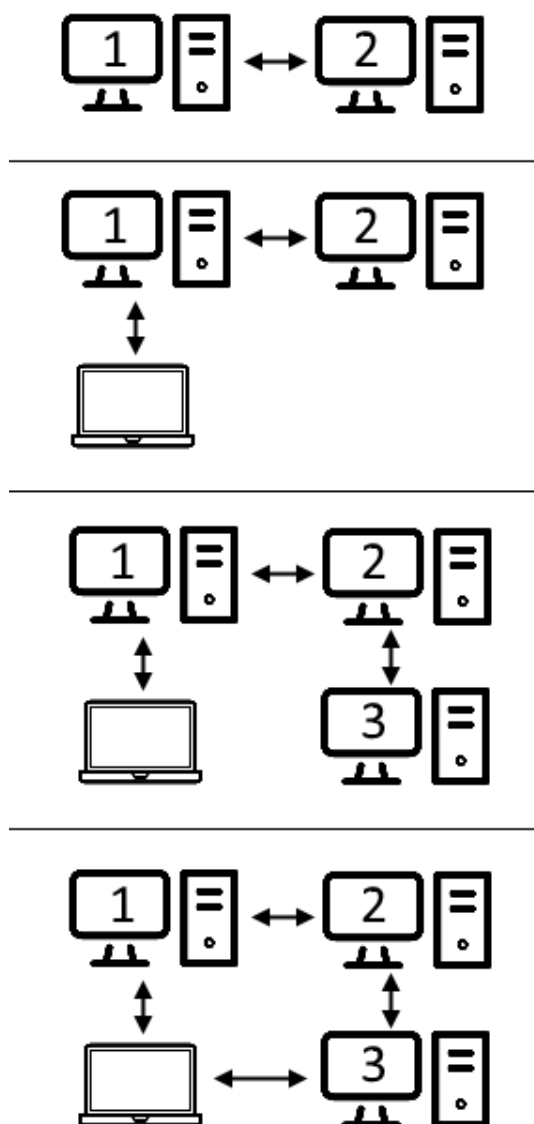


Рисунок 2.5 Процесс подключения нового мастер-узла

Из спроектированной архитектуры следует, что доступ к информации для новых клиентов сохраняется до тех пор, пока в сети находится хотя бы один мастер-узел.

2.4. Транзакции

В сети blockchain возможны коллизии. Для их предотвращения используются транзакции. Транзакция – это данные, которые пользователь добавляет в сеть. В транзакции также содержится различная служебная информация. Все блоки состоят из транзакций. Если транзакция ещё не находится в блоке, она считается неподтверждённой.

Представим, что вместо транзакции отправитель сразу подписывает блок и отправляет его. Тогда может произойти следующая ситуация. Два узла одновременно отправляют новую информацию в сеть, при этом они подключены к разным мастер-узлам. Тогда у двух разных мастер-узлов в одном по счёту блоке окажется разная информация, при этом оба блока будут валидны. Соответственно, произойдёт разделение сети на две разных цепочки. Именно поэтому вместо блоков необходимо отправлять транзакции.

В процессе проектирования реестра было принято решение, что транзакции будут хранить следующие данные:

1. Временная метка на момент создания (количество миллисекунд, прошедших с 1 января 1970 года).
2. Тип транзакции.
3. Сама передаваемая информация.
4. Публичный ключ отправителя.
5. Хеш транзакции.
6. Цифровая подпись отправителя.

Разберём подробнее для чего используется каждый пункт.

В реестре запрещены одинаковые транзакции, так как иначе злоумышленник в любой момент может отправить ещё раз транзакцию, которая уже есть в цепочке, потому что она имеет верную подпись. Однако из-за такого ограничения появляется другая проблема. Самому отправителю всё же может потребоваться добавить в реестр точно такие же данные. Именно для

этого необходимо указывать время создания. Оно используется для уникализации транзакции. Отправитель в любой момент сможет подписать новую транзакцию с уже имеющимися в цепочке данными, а у злоумышленника такой возможности нет. Однако использовать данное поле для проверки времени добавления записи в реестр нельзя, так как отправитель может указать недостоверное время.

Тип транзакций используется для того, чтобы отличать транзакции с информацией от служебных.

С помощью публичного ключа можно удостовериться, что данный участник имеет право добавлять новые данные.

Хеш транзакции нужен для вычисления цифровой подписи, а также для однозначной идентификации транзакции.

Подпись позволяет установить, что указанный публичный ключ действительно принадлежит отправителю, а информация в транзакции не была изменена злоумышленником.

Из сказанного выше можно сделать выводы, что обычные узлы должны только отправлять транзакции, а собирать их в блоки должен кто-то другой. Однако если предоставить эти полномочия всем мастер-узлам возникнет проблема, описанная в следующем пункте.

2.5. Алгоритм консенсуса

Представим такую ситуацию: два пользователя одновременно отправляют транзакции в сеть. При этом они подключены к разным мастер-узлам, между которыми присутствует некая задержка в сети. В таком случае один мастер-узел добавит в блок одну транзакцию, а второй другую. То есть опять же произойдёт разделение цепочки блоков на две ветки. Для решения этой проблемы в blockchain проектах используется алгоритм консенсуса [14].

В общем понимании консенсус – это способ прийти к какому-либо соглашению. В blockchain алгоритм консенсуса необходим, чтобы быть уверенным в том, что лицо, добавившее новый блок, сделало это правомерно.

Соответственно, для обеспечения работоспособности все участники сети должны соблюдать этот алгоритм.

Самым используемым алгоритмом достижения консенсуса является алгоритм доказательства выполнения работы Proof of Work (PoW). Данный алгоритм используется в сетях Bitcoin, Litecoin, Ethereum и ещё множестве известных проектов. Его суть заключается в решении очень сложной математической задачи (майнинге). Так как на нахождение ответа требуется потратить большое количество времени, вероятность того, что два мастер-узла найдут его одновременно, крайне мала. Соответственно, узлы принимают только блоки от мастер-узлов, которые предоставляют правильные ответы. Несмотря на популярность, у такого подхода присутствуют существенные недостатки. Первый из них – атака 51%. В случае, если у атакующей стороны находится более 50% всей вычислительной мощности, она имеет возможность выбирать, какие блоки подтверждать, а какие нет. Вторым минусом PoW заключается в том, что для решения математических задач требуется огромное количество электричества, при этом результаты решения не являются полезными и нигде больше не применяются. Соответственно, большое количество электроэнергии расходуется впустую. Ещё один недостаток PoW – скорость подтверждения блоков. Чем больше мы хотим снизить вероятность одновременного нахождения решения, тем больше времени на его нахождение должно уходить.

Вторым из наиболее используемых механизмов консенсуса – алгоритм доказательства владения Proof of Stake (PoS). Используется в BlackCoin, Nxt и других сетях. Данный алгоритм был придуман для замены Proof of Work. При использовании PoS сходит на нет проблема впустую потраченной электроэнергии, потому как мастер-узел, который будет подписывать блок, выбирается, исходя из баланса на счету. Соответственно в таком случае огромные вычислительные мощности не

требуются. Однако для использования в нашем реестре такой вариант не подходит, потому что механизм Proof of Stake завязан на криптовалюте.

Для достижения консенсуса в разрабатываемом реестре был выбран алгоритм доказательства полномочий Proof of Authority (PoA). Данный алгоритм подразумевает наличие ограниченного числа валидаторов блоков. То есть подтверждать блоки могут только заранее утверждённые мастер-узлы. Данный механизм является энергоэффективным и при этом обеспечивает высокую скорость добавления новых блоков, в отличие от Proof of Work.

Валидаторы – это отдельный класс мастер-узлов. Валидаторы являются самыми важными узлами в сети, так как формируют из новых транзакций блоки и подписывают их. С точки зрения сети они почти ничем не отличаются от остальных мастер-узлов. Однако для них существуют отдельные ограничения и требования. Во-первых, максимальное их количество в сети ограничено и задаётся при создании новой цепочки блоков. Во-вторых, в отличие от мастер-узлов, которые в любой момент могут отключиться от сети, валидаторы обязаны всегда быть доступны, так как добавляют новые блоки в сеть. Чем меньше валидаторов доступно, тем дольше добавляются блоки. Добавление информации в реестр возможно, пока хотя бы один валидатор находится в сети.

Однако проблема, которую мы пытались решить внедрением алгоритма консенсуса всё ещё никуда не пропала – для обеспечения децентрализации в сети должно находиться несколько валидаторов. Соответственно, разделение цепочки блоков может произойти и в этом случае.

Значит, необходимо создать алгоритм, который решает, какой из валидаторов может подписывать блок на данный момент. Один из вариантов – выбирать валидатора исходя из номера текущего блока. Однако у данного подхода есть один существенный недостаток – если от сети отключится валидатор, который должен подписывать текущий блок, этот блок больше никто не сможет подписать, значит добавление новой информации

станет невозможным. Соответственно, выбирать валидатора необходимо исходя из текущего времени. Окончательным вариантом решения данной проблемы стал следующий алгоритм:

1. Фиксируем текущую временную метку (количество миллисекунд, прошедших с 1 января 1970 года).
2. Делим его на время валидации.
3. Округляем до ближайшего целого.
4. Берём результат пункта 3 по модулю числа валидаторов.
5. Получившийся результат будет номером валидатора, который может подписать блок в данный момент (нумерация от нуля).

Время валидации здесь показывает, как долго один валидатор может подписывать блоки во время своей очереди.

Рассмотрим на примере работу алгоритма при количестве валидаторов равном 3 и времени валидации равном 2 секундам (на практике этот параметр должен быть больше). Результаты приведены в таблице 2.2.

Таблица 2.2 – Пример работы алгоритма

Временная метка, мс	Время GMT+03:00	Валидатор
1590266468000	20:41:08	1
1590266469000	20:41:09	2
1590266470000	20:41:10	2
1590266471000	20:41:11	0
1590266472000	20:41:12	0
1590266473000	20:41:13	1
1590266474000	20:41:14	1

Как мы видим, валидаторы меняются каждые две секунды по кругу.

Данный алгоритм будет применяться в реестре следующим образом:

1. Валидаторы применяют алгоритм и проверяют, их ли время подтверждать блоки.
2. Если да, то валидатор формирует и подписывает блок, при этом указывая время подписи.

3. Во время своей очереди валидатор может сделать это неограниченное количество раз, но только если ещё осталась неподтверждённая информация.

4. При получении нового блока все узлы применяют алгоритм к времени, которое указано в блоке, и проверяют мог ли этот валидатор подписывать блоки в это время. Узлы так же проверяют больше ли время в новом блоке, чем в предыдущем. К тому же время в новом блоке не должно быть больше текущего. Так как время не может быть настроено идеально, данный пункт проверяется с некоторой погрешностью, которая должна быть не больше, чем разность времени полного круга валидации и времени валидации для одного валидатора).

5. Если все пункты соблюдены, значит блок подписал верный валидатор.

Дополнительные проверки времени необходимы, так как при подписи блока валидатор может указать любое время, в том числе недостоверное.

Остаётся лишь определить, кто является валидатором, а кто нет. Информацию об этом предлагается хранить в самом первом блоке.

2.6. Генезис-блок

Самый первый блок в цепочке принято называть генезис-блоком. Обычно этот блок отличается от всех остальных блоков хотя бы потому, что хеш предыдущего блока в этом случае вычислить невозможно.

При проектировании реестра было принято решение включить в генезис-блок следующие данные:

1. Время создания генезис-блока.
2. Генезис-транзакция.
3. Хеш генезис-блока.

Наличие транзакции в этом случае необходимо только для совместимости с другими блоками. В данном случае информацию можно было бы добавить напрямую в блок.

Генезис-транзакция включает в себя следующие данные:

1. Тип транзакции.
2. Открытые ключи валидаторов.
3. Открытые ключи отправителей (людей, имеющих право на добавление информации).

Тип транзакции помогает отличить генезис транзакцию от всех остальных транзакций.

Когда поступает новый блок, узлы обращаются к генезис-блоку и проверяют, находится ли ключ валидатора, подписавшего этот блок, в генезис-транзакции.

Когда поступает новый блок или транзакция, узлы смотрят, находится ли в генезис-транзакции публичный ключ отправителя, чтобы удостовериться, что транзакция отправлена участником, который имеет на это право.

Генезис-блок – единственный блок, который обычно идёт в поставке с программным обеспечением blockchain проектов. После сверки этого блока с мастер-узлами узлы загружают остальные блоки уже с этих мастер-узлов. Именно с помощью него узлы могут понять, что подключились к нужной сети. Так, если изменить генезис-блок, то, по причине изменения хеша, остальные блоки тоже не будут совпадать. Соответственно, в этом случае это будет другая сеть, к которой смогут подключиться те, у кого есть соответствующий генезис-блок.

2.7. Структура блоков

Теперь рассмотрим структуру всех остальных блоков. При проектировании реестра было принято решение хранить в блоках следующие данные:

1. Время валидации блока.
2. Транзакции.
3. Публичный ключ валидатора.
4. Хеш предыдущего блока.

5. Хеш данного блока.
6. Цифровая подпись валидатора.

Разберём подробнее для чего нужны каждые данные.

Время валидации блока необходимо для работы алгоритма консенсуса, о котором было рассказано ранее. Именно это время следует использовать для проверки времени добавления записи в реестр.

Транзакций в блоке может быть неограниченное количество.

Публичный ключ валидатора используется для работы механизма консенсуса. Он сверяется с ключами, которые находятся в генезис-блоке.

Хеш предыдущего блока используется для обеспечения правильной последовательности блоков.

Хеш данного блока используется для обеспечения правильной последовательности блоков, однозначной идентификации блока и создания цифровой подписи.

Подпись позволяет установить, что указанный публичный ключ действительно принадлежит валидатору, а информация в блоке не была изменена злоумышленником.

2.8. Добавление новых данных в реестр

Из предыдущих пунктов мы узнали, как устроены блоки, для чего нужны валидаторы, транзакции и алгоритм консенсуса. Теперь можно подвести итог и по пунктам расписать, как же происходит добавление новой информации в реестр:

1. Узел добавляет в транзакцию данные, которые хочет внести в реестр, подписывает её и отправляет всем доступным мастер-узлам.

2. Мастер-узлы проверяют хеш транзакции, публичный ключ отправителя, цифровую подпись, нет ли такой же транзакции в цепочке или пуле неподтверждённых транзакций. Если все данные верны, они заносят транзакцию в этот пул. Данный пул, как и цепочка блоков, синхронизируется

между всеми участниками сети. В нём хранятся все транзакции, которые ещё не попали в блок.

3. Если среди тех мастер-узлов не оказалось валидаторов по причине отсутствия прямого соединения с отправителем, они получают эту транзакцию через мастер-узлы.

4. Валидаторы, как и все остальные узлы, проверяют транзакцию по параметрам, приведённым выше, и заносят её в пул неподтверждённых транзакций.

5. Валидатор, который согласно алгоритму консенсуса в данный момент может подтверждать блоки, формирует блок из транзакций, которые находятся в пуле неподтверждённых транзакций, подписывает его и добавляет в цепочку. При этом он удаляет из пула все внесённые в блок транзакции.

6. Все остальные узлы во время синхронизации получают этот новый блок. После чего они проверяют на валидность все транзакции в этом блоке, соответствие алгоритму консенсуса, открытый ключ валидатора, хеш этого блока, соответствие хеша предыдущего блока, указанного в блоке, и хеша последнего блока в цепочке, цифровую подпись валидатора. Если все эти параметры верны, узлы заносят в цепочку новый блок, при этом удаляя из пула неподтверждённых транзакций все транзакции, находящиеся в этом блоке.

Теперь, когда мы рассмотрели принципы работы blockchain, сравнили и выбрали технологии, рассмотрели и спроектировали структуру публичного децентрализованного реестра, составили алгоритмы его работы, можно приступить к разработке программного обеспечения для создания такого реестра и работы с ним.

3. РАЗРАБОТКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ДЛЯ ОРГАНИЗАЦИИ РЕЕСТРА

3.1. Выбор средств разработки

Для разработки любого программного обеспечения необходимо сначала выбрать средства разработки. В нашем случае они должны удовлетворять следующим требованиям:

1. Асинхронность. Так как в децентрализованных системах производится одновременная работа со множеством входящих и исходящих соединений, язык программирования должен поддерживать асинхронность. Это значит, что пока мы ждём ответа от какого-либо мастер узла, остальная часть приложения должна продолжать выполнять свои функции.

2. Возможность создания кроссплатформенного пользовательского интерфейса. Полученная программа должна иметь пользовательский интерфейс и запускаться в браузерах и на операционных системах Linux и Windows.

В итоге для разработки программного обеспечения был выбран язык JavaScript, среда исполнения Node.js и фреймворк Electron. Для разработки использован редактор Visual Studio Code. Разберём каждый пункт подробнее.

JavaScript – язык программирования, который обычно применяется при разработке веб-страниц и запускается в браузере. Так как JS исторически применялся в браузерах, он обычно работал с подключениями к серверу, а значит ему была необходима асинхронность. Благодаря этому сейчас в JavaScript по умолчанию присутствует удобная работа с асинхронными функциями. К тому же подавляющее большинство стандартных функций и сторонних библиотек написаны с применением асинхронного подхода.

Как было сказано выше, JavaScript – это в основном браузерный язык, однако с помощью него мы можем писать и приложения для Windows, Linux, MacOS и других операционных систем. Именно для этого мы используем Node.js. Node.js – это среда исполнения, которая заменяет браузер и позволяет

языку выполняться из операционной системы. К тому же она позволяет взаимодействовать с помощью JS с внешними устройствами, файловой системой и API операционной системы. Ещё одним преимуществом использования JavaScript и Node.js является пакетный менеджер npm, который содержит большое количество готовых к использованию библиотек, предоставляет документацию и их удобную установку. Например, npm позволяет с лёгкостью найти множество библиотек для работы со строками, криптографией и HTTP-запросами.

Редактор Visual Studio Code был выбран, потому что является бесплатным, имеет встроенный интерфейс для работы с Git и GitHub, а также предоставляет удобную систему плагинов, с помощью которой можно легко найти и установить плагины, упрощающие разработку. Например, автодополнение, линтер, который позволяет писать качественный код, и другие.

Вся разработка велась с использованием системы контроля версий Git. Работа доступна в GitHub репозитории GVal98/blockchain-registry.

3.2. Разрабатываемые приложения

Для большей наглядности было принято решение показать разработку приложений на примере реестра недвижимости. Наш реестр будет содержать следующую информацию:

1. Идентификатор недвижимости.
2. Идентификатор продавца.
3. Идентификатор покупателя.
4. Сумма сделки.

В качестве идентификатора недвижимости будем использовать кадастровый номер, для идентификации продавца и покупателя – их ИНН.

Было решено разработать три приложения: консольное приложение, приложение с пользовательским интерфейсом и тонкий веб-клиент. Они будут

применяться для решения разных типов задач и иметь различный функционал. Сравнение этих приложений приведено в таблице 3.1.

Таблица 3.1 – Сравнение разрабатываемых приложений

Интерфейс	Консольный	Графический	Веб
Поддерживаемые операционные системы	Windows, Linux	Windows, Linux	Windows, Linux, MacOS, Android, iOS и другие платформы с наличием веб-браузера
Хранение всей цепочки блоков	Да	Да	Нет
Проверка блоков и транзакций	Да	Да	Нет
Взаимодействие с сетью	Со всеми мастер-узлами	Со всеми мастер-узлами	С одним мастер-узлом
Запуск обычного узла	Нет	Да	Нет
Запуск мастер-узла	Да	Да	Нет
Запуск узла-валидатора	Да	Нет	Нет
Запуск веб-интерфейса	Да	Нет	Нет
Отправка транзакций	Нет	Да	Да
Удобный просмотр транзакций	Нет	Да	Да

Разделение на консольное и графическое приложение необходимо, потому что валидаторы должны быть в сети постоянно, соответственно узлы валидации не должны запускаться через графический интерфейс пользователя, а должны быть запущены в консольном режиме на сервере. Также имеется возможность запустить в консольном режиме мастер-узел для повышения надёжности сети. В консольном приложении нельзя удобно просматривать транзакции и создавать их, по этой причине его нельзя запустить как обычный узел.

Соответственно, через графическое приложение нельзя запустить узел валидации, потому что надёжность сети от этого уменьшится. Но с его помощью можно запустить мастер-узел, так как чем больше в сети мастер-

узлов, тем лучше. При этом основная задача, которую решает приложение с графическим интерфейсом – запуск обычного узла для удобного просмотра и добавления транзакций.

Главная особенность веб-приложения заключается в том, что оно не хранит все блоки, а загружает нужные с доверенного мастер-узла.

3.3. Структура консольного приложения

В первую очередь было решено разработать консольное приложение. Для разработки приложения на Node.js необходимо инициализировать новый проект с помощью команды `npm init`. Далее необходимо будет указать автора, название и версию приложения, описание, лицензию и репозиторий. Теперь можно приступить непосредственно к разработке.

Получившаяся в результате разработки структура приложения представлена в таблице 3.2

Таблица 3.2 – структура консольного приложения

Файл	Описание
app.js	Точка входа в приложение. Инициализирует весь функционал.
block-helper.js	Отвечает за операции с блоками. Создание и валидацию.
blockchain-handler.js	Blockchain менеджер. Инициализирует процессы обновления, загрузки и сохранения цепочки блоков.
blockchain.json	Хранит цепочку блоков по умолчанию.
connection-handler.js	Менеджер подключений. Отвечает за сетевое взаимодействие, получение доступных узлов, новых транзакций, загрузку новых блоков.
elliptic.js	Отвечает за криптографию.
LICENSE	Файл лицензии GNU GPL v3. Создан GitHub автоматически.
nodes.json	Хранит мастер-узлы по умолчанию.
package-lock.json	Хранит используемые приложением зависимости и их версии.
package.json	Хранит данные о приложении, заданные при его инициализации с помощью <code>npm init</code> , скрипты запуска и зависимости.
request.js	Отвечает за отправку HTTP-запросов.

Продолжение таблицы 3.2

server.js	HTTP-сервер. Отвечает за функционал мастер-узла, обработку и ответы на входящие запросы.
transaction-helper.js	Отвечает за операции с транзакциями. Создание и валидацию
web	Папка, содержащая ресурсы для веб-версии

Каждый .js файл содержит соответствующий класс, который отвечает за перечисленные функции. Инициализация консольного приложения происходит следующим образом:

1. Запускается точка хода в приложение, в которой подключаются все необходимые для старта модули.
2. Приложение запускает менеджер подключений.
3. Приложение ждёт, когда менеджер подключений подключится к мастер-узлам.
4. После этого приложение запускает blockchain менеджер.
5. Приложение ждёт, когда blockchain менеджер загрузит из файла уже имеющуюся цепочку блоков.
6. После этого запускается HTTP-сервер, то есть функционал мастер-узла.

Теперь разберём подробнее каждую часть приложения.

3.4. Формат данных

Для разработки было необходимо определиться, какой использовать формат для передачи и хранения данных. В итоге был выбран формат JSON (JavaScript Object Notation).

JSON – это текстовый формат для обмена данными. В формате JSON возможно хранить два типа структур: пары ключ-значение и массивы данных. Ключом может быть только строка, а значением может быть другая пара ключ-значение, массив, число, логические значения.

Наряду с XML, JSON является самым используемым форматом данных для хранения данных, особенно в веб-приложениях и при передаче по сети.

Преимуществами JSON по сравнению с XML являются простота и скорость парсинга данных, так как изначально формат JSON произошёл именно от JavaScript. К тому же в JSON хранится гораздо меньше служебной информации, так как в XML используется система тэгов, в которой имя ключа дублируется. Соответственно, одни и те же данные в формате JSON будут использовать гораздо меньше памяти и пропускной способности, чем в формате XML.

Далее были определены названия, типы переменных и сама структура JSON.

Пример транзакции в формате JSON показан на рисунке 3.1.

```
{
  "time": 1589999187041,
  "type": "data",
  "data": {
    "property": "46:11:105983:810",
    "seller": 7707291958,
    "buyer": 7707521112,
    "price": 5483900
  },
  "sender": "475727ef9e6ec139bd0e5b5022afa0eecacabbbcacd586aa0fe839b38b201027",
  "hash": "15ea573f2a4b557df0f2e966ee38b7975c0e0e998975f35137d1e253867a6d96",
  "sign": "9BB209E2110EA6BA6CBE93E3AB949BFE3BB537FC0A519A5EE5FA349606EC539BC389
  B91977ABC24335897E5F5573B2667B72F5CF1E4F2B555BE3183323177209"
}
```

Рисунок 3.1 – пример транзакции в формате JSON

Формат и описание каждой пары ключ-значение приведены в таблице 3.3.

Для чего используется те или иные данные в транзакции, подробнее расписано в пункте 2.4 работы.

Таблица 3.3 – описание пар ключ-значение транзакций

Ключ	Тип	Описание
time	Число	Время транзакции в миллисекундах
type	Строка	Тип транзакции. Обычно равен “data”
data	Любой. В нашем случае объект	Отправляемые данные. Могут быть абсолютно любой структуры и формата в зависимости от реестра
sender	Строка	Публичный ключ отправителя

Продолжение таблицы 3.3

hash	Строка	Хеш транзакции
sign	Строка	Цифровая подпись отправителя

В таблице 3.4 приведено описание хранимых в объекте data пар ключ-значение, которые применяются именно для разрабатываемого реестра недвижимости.

Таблица 3.4 – пары ключ-значение объекта данных

Ключ	Тип	Описание
property	Строка	Кадастровый номер
seller	Число	ИНН продавца
buyer	Число	ИНН покупателя
price	Число	Сумма сделки

Пример блока в формате JSON с двумя транзакциями показан на рисунке 3.2.

Формат и описание каждой пары ключ-значение приведены в таблице 3.5.

Таблица 3.5 – описание пар ключ-значение блоков

Ключ	Тип	Описание
time	Число	Время подписи блока в миллисекундах
transactions	Массив	Массив, хранящий объекты транзакций
validator	Строка	Открытый ключ валидатора
previousHash	Строка	Хеш предыдущего блока
hash	Строка	Хеш данного блока
sign	Строка	Цифровая подпись валидатора

Для чего используется та или иная информация в блоке, подробнее расписано в пункте 2.7.

Пример генезис-блока с тремя валидаторами и тремя отправителями в формате JSON показан на рисунке 3.3.

Формат и описание каждой пары ключ-значение приведены в таблице 3.6.

Подробнее о генезис-блоке написано в пункте 2.6 работы.

```

{
  "time": 1590421644379,
  "transactions": [
    {
      "time": 1590421629252,
      "type": "data",
      "data": {
        "property": "44:14:109169:803",
        "seller": 7708152232,
        "buyer": 7708912874,
        "price": 5465200
      },
      "sender": "475727ef9e6ec139bd0e5b5022afa0eecacabbbcacd586aa0fe839b38b201027",
      "hash": "2771a83c7a834fb0c70b85e64a1158422b0f83e67dbea144da1ac69fa1b01ffa",
      "sign": "D48DCA7E23233DEFA5709F5F620B1638E37F26F99CB4E514E8B3243A70B4850203CEF9
958D007B35BD4A8C5532F707C05C8914F9D0801F6B0719904ECF42EA06"
    },
    {
      "time": 1590421629310,
      "type": "data",
      "data": {
        "property": "47:12:100019:807",
        "seller": 7708315262,
        "buyer": 7709216720,
        "price": 1974083
      },
      "sender": "475727ef9e6ec139bd0e5b5022afa0eecacabbbcacd586aa0fe839b38b201027",
      "hash": "f4b7e8c4102ce8ab18cde1217160ec2b8a8bd483896b7e43ff60ad6ffff9a5a63",
      "sign": "DACC58C2121EBC2C5FCA7318543538518F287741875CB65151C1676A8CDBC7015345C2
82A6CEA50B8D6CD1C23EC347663419F5A20D59D945A4CB99A4B492340A"
    }
  ],
  "validator": "dd0ee245b9ff7c859710a6e83b10108ac59796c7c736e918117146b601471f21",
  "previousHash": "9dc4c9c90722fd68baef1288fef0bcc985258eb5a66b16c12c24cd16afa947e0",
  "hash": "b94bc51586437e87b6d4c5e41a4b1f9ea39610d13bffa19ce8e07c7a932d4a7",
  "sign": "2C2F0DE6E9EE6886516AE2F1A232E4FAC41CE6E925087CCD15C62E25247F925B9DA08AD018
37C78B6B8B3FB6CAEC6041C89BF3D99FD765A1BC0A481BDCE60808"
}

```

Рисунок 3.2 – пример блока в формате JSON

Таблица 3.6 – описание пар ключ-значение генезис-блока

Ключ	Тип	Описание
time	Число	Время создания блока в миллисекундах
transactions	Массив	Массив, хранящий объект генезис-транзакции
type	Строка	Тип транзакции. В этом случае равен “genesis”
data	Объект	Содержит массивы validator и senders
validators	Массив	Содержит публичные ключи валидаторов
senders	Массив	Содержит публичные ключи отправителей

```

{
  "time": 1589998438479,
  "transactions": [
    {
      "type": "genesis",
      "data": {
        "validators": [
          "cfab668b139b5e6c056d336cdbe064f98ed4057dfc2389cdc19eb8d7305cf776",
          "e61e60f16436063d923e2325f8a6e758085e1b4164fa051a7a13dadc74741380",
          "dd0ee245b9ff7c859710a6e83b10108ac59796c7c736e918117146b601471f21"
        ],
        "senders": [
          "1e6c708a24317ec06bf92cf4bc100e0532d78a6e6d100e6116fdc09404363b40",
          "a973e616256036a7589ef05f77de0d10c48280f4be2f6323e6148ea290674ddd",
          "475727ef9e6ec139bd0e5b5022afa0eecacabbbcacd586aa0fe839b38b201027"
        ]
      }
    ]
  ],
  "hash": "9dc4c9c90722fd68baef1288fef0bcc985258eb5a66b16c12c24cd16afa947e0"
}

```

Рисунок 3.3 – пример генезис-блока в формате JSON

3.5. Реализация сетевого взаимодействия

3.5.1. Выбор протокола передачи данных

В первую очередь было решено реализовать сетевое взаимодействие между узлами. Для начала необходимо выбрать сетевой протокол.

В качестве протокола передачи данных был выбран протокол HTTPS. HTTP – это протокол прикладного уровня, который работает поверх TCP. В HTTPS данные в целях безопасности шифруются с помощью TLS.

HTTP-запрос состоит из следующих данных:

1. Начальная строка, которая указывает метод запроса (GET, POST и другие) и путь до запрашиваемых данных (URI).
2. Заголовки, которые отвечают за параметры передачи, например кодировку, язык, тип данных.
3. Тело запроса, в котором передаётся само сообщение.

Ответ же состоит из служебных заголовков и самого тела ответа.

Несмотря на то, что протокол HTTP отправляет довольно много служебной информации, он создаёт дополнительный слой абстракции над TCP, тем самым обеспечивая следующие преимущества:

1. Удобная маршрутизация с помощью URI.
2. Решение проблем с сегментацией TCP пакетов.
3. Решение проблем с кодировками и типом передаваемых данных.
4. Простота разработки – имеется большое количество библиотек как для создания HTTP-сервера, так и для отправки HTTP-запросов.
5. Протокол поддерживается всеми браузерами, что позволяет реализовать веб-версию приложения.
6. HTTPS из коробки поддерживает шифрование с помощью TLS.

Приложение будет иметь возможность как генерировать самоподписанные сертификаты, так и использовать готовые.

Для отправки HTTP-запросов используется библиотека `needle`. Для создания HTTP-сервера используется фреймворк `express`.

3.5.2. Проектирование API

Теперь, когда протокол передачи данных выбран, необходимо спроектировать API (программный интерфейс приложения). API – это набор правил и инструкций, по которым узлы будут взаимодействовать друг с другом.

Обмениваться информацией узлы будут по следующему принципу:

1. Узел отправляет мастер-узлу POST запрос. При этом он указывает команду, которую должен выполнить мастер-узел в URI (например, запросить какой-либо блок). Параметры этого запроса передаются в теле запроса в формате JSON (например, номер запрашиваемого блока).
2. Мастер-узел присылает ответ в формате JSON.

Далее будут приведены все команды, которые узлы будут поддерживать, и их описание.

URI: /getBlock

Описание: возвращает блок указанной высоты.

Параметры: высота (порядковый номер в цепочке) блока.

На рисунке 3.4 показан пример тела запроса.

```
{  
  "blockHeight": 3  
}
```

Рисунок 3.4 Пример тела запроса /getBlock

На рисунке 3.5 показан пример тела ответа

```
{  
  "result": {  
    "time": 1590425111224,  
    "transactions": [  
      {  
        "time": 1590425101131,  
        "type": "data",  
        "data": {  
          "property": "40:10:102887:807",  
          "seller": 7707128655,  
          "buyer": 7710043696,  
          "price": 4147630  
        },  
        "sender": "475727ef9e6ec139bd0e5b5022afa0eecacabbbbcacd586aa0fe839b38b201027",  
        "hash": "1e979f316f344da4d772eba36abadaab123ad7fa131719509927b7f2857f7524",  
        "sign": "EFC7E3A742C0B23031F70AE5A61DF05FB8AD9E994C27851A68B422B379D156F43E9D3A  
B049E67849A5952EDB0E34B2D10C57C63B75FA4D3F8D6A92053397BC05"  
      },  
      ],  
    "validator": "dd0ee245b9ff7c859710a6e83b10108ac59796c7c736e918117146b601471f21",  
    "previousHash": "69e7fa06cfd650b0e7ecce7dd74331d0378878f038ec2402e27a8fd72a3783ef",  
    "hash": "928e7bc845489e81a992707ec4e106204d2c32bf9e597d0b5a545c58a7b4cfad",  
    "sign": "3422521B428F981866F68A18868430F367665243EEDB5B419E4C66FB8DA087BB9CBDA10B4B  
DB08BC75E232F7CC17A4044B739FD049C74588332264F37ECCB80A"  
  }  
}
```

Рисунок 3.5 Пример тела ответа /getBlock

URI: /getBlocks

Описание: возвращает несколько блоков

Параметры: высота начального и конечного блока

На рисунке 3.6 показан пример тела запроса

```
{
  "startBlock": 0,
  "endBlock": 1
}
```

Рисунок 3.6 Пример тела запроса /getBlocks

На рисунке 3.7 показан пример тела ответа

```
{
  "result": [
    {
      "time": 1589998438479,
      "transactions": [
        {
          "type": "genesis",
          "data": {
            "validators": [
              "cfab668b139b5e6c056d336cde064f98ed4057dfc2389cdc19eb8d7305cf776",
              "e61e60f16436063d923e2325f8a6e758085e1b4164fa051a7a13dadcd74741380",
              "dd0ee245b9ff7c859710a6e83b10108ac59796c7c736e918117146b601471f21"
            ],
            "senders": [
              "1e6c708a24317ec06bf92cf4bc100e0532d78a6e6d100e6116fdc09404363b40",
              "a973e616256036a7589ef05f77de0d10c48280f4be2f6323e6148ea290674ddd",
              "475727ef9e6ec139bd0e5b5022afa0eecaabbbcacd586aa0fe839b38b201027"
            ]
          }
        }
      ],
      "hash": "9dc4c9c90722fd68baef1288fef0bcc985258eb5a66b16c12c24cd16afa947e0"
    },
    {
      "time": 1590426279510,
      "transactions": [
        {
          "time": 1590426264423,
          "type": "data",
          "data": {
            "property": "46:14:118278:804",
            "seller": 7708763686,
            "buyer": 7707909012,
            "price": 2528143
          },
          "sender": "475727ef9e6ec139bd0e5b5022afa0eecaabbbcacd586aa0fe839b38b201027",
          "hash": "c5d6cb8c34c3ef7d211b0c6b30e1efd728caed783ce7fafb23a075b73fb60db2",
          "sign": "EFF66410E745B00660DAD92037CD0A8009C0CE49EC88B5CEF0742A493B04DA0EDB868D0299E4F9E76A22546C5D43CD510093CEE1F7F658579A80C74B38FA1403"
        }
      ],
      "validator": "dd0ee245b9ff7c859710a6e83b10108ac59796c7c736e918117146b601471f21",
      "previousHash": "9dc4c9c90722fd68baef1288fef0bcc985258eb5a66b16c12c24cd16afa947e0",
      "hash": "6f7faa689a1d815e9c81619eb8f8e9ad932866bcd5147438a4c2f5c354f85832",
      "sign": "A29692A66CB710E7B3D67230DAD66501AB266276EE2B39311A2A023FD12D8136220FF0E1DD0DD60A178F3F27FFF524271D184F3C59E2642BED00F39E4B812900"
    }
  ]
}
```

Рисунок 3.7 Пример тела ответа /getBlocks

URI: /getPendingTransactions

Описание: возвращает высоту цепочки блоков мастер-узла и его пул неподтверждённых транзакций.

Параметры: отсутствуют.

На рисунке 3.8 показан пример тела ответа.

```
{
  "result": {
    "height": 4,
    "pendingTransactions": [
      {
        "time": 1590428053618,
        "type": "data",
        "data": {
          "property": "42:10:106280:809",
          "seller": 7708916777,
          "buyer": 7707685810,
          "price": 1369490
        },
        "sender": "475727ef9e6ec139bd0e5b5022afa0eecacabbbcacd586aa0fe839b38b201027",
        "hash": "5dad782775656e61c7f867ddc3baee8ef9a12ee517f0444695b18d153f15cf63",
        "sign": "272E3671FEDCF04EBBF5E9111361212939FD7FE9FB2FA6F1FEACECB3198855E912BD91E27D356E2561CCAF32AE48D887959F893E0E4128D485F403546C608"
      },
      {
        "time": 1590428053682,
        "type": "data",
        "data": {
          "property": "44:12:100786:811",
          "seller": 7707717280,
          "buyer": 7708064897,
          "price": 7363616
        },
        "sender": "475727ef9e6ec139bd0e5b5022afa0eecacabbbcacd586aa0fe839b38b201027",
        "hash": "5053ae339fc7c82f562f6cb4826a2ff13434aa0369e5f02e1c5a6e520abadd99",
        "sign": "2F16967B88A1CEE471B6CF0E968D1142776567AAE3FCF9E6DB98C634DA1C3FAAE83DEAC1B36534814084C90E589FE18FFF2012D486A6F96C9EC073C5308AAC06"
      }
    ]
  }
}
```

Рисунок 3.8 Пример тела ответа /getPendingTransactions

URI: /sendTransaction

Описание: отправляет новую транзакцию мастер-узлу. Возвращает результат проверки транзакции мастер-узлом.

Параметры: транзакция.

На рисунке 3.9 показан пример тела запроса

```
{
  "time": 1590429412005,
  "type": "data",
  "data": {
    "property": "42:12:109511:804",
    "seller": 7709265454,
    "buyer": 7709173480,
    "price": 1055552
  },
  "sender": "475727ef9e6ec139bd0e5b5022afa0eecacabbbcacd586aa0fe839b38b201027",
  "hash": "321af32529f28a815e37f603e55fafb980570d323807f2551b13890e54f93012",
  "sign": "E186E8BCD208E41DAD67BF05AF705595890F81BDADB6A96AF881DDB403BC4EBA7C34743DC5015801CFCA11E873543BD7D80897BE5CECB0D38E99580D7188F02"
}
```

Рисунок 3.9 Пример тела запроса /sendTransaction

На рисунке 3.10 показан пример тела ответа.

```
{
  "result": true
}
```

Рисунок 3.10 Пример тела ответа /sendTransaction

URI: /getNodes

Описание: возвращает высоту цепочки блоков мастер-узла, его пул неподтверждённых транзакций, все известные ему мастер-узлы. Позволяет новому мастер-узлу объявить сети свой адрес.

Параметры: адрес мастер-узла, с которого отправляется запрос. Тело запроса будет пустым, если он посылается от обычного узла.

На рисунке 3.11 показан пример тела запроса.

```
{
  "ip": "180.240.170.150",
  "port": 1111
}
```

Рисунок 3.11 Пример тела запроса /sendTransaction

На рисунке 3.12 показан пример тела ответа.


```

{
  "result": {
    "height": 4,
    "pendingTransactions": [
      {
        "time": 1590431173076,
        "type": "data",
        "data": {
          "property": "45:10:102441:805",
          "seller": 7708547072,
          "buyer": 7708285218,
          "price": 9280395
        },
        "sender": "a973e616256036a7589ef05f77de0d10c48280f4be2f6323e6148ea290674ddd",
        "hash": "870d3ca313621d88fcfbe209d1098088ce65d4b93c38f98edbeb6379df0c37c0",
        "sign": "3F129888D1A03DAC998452386F1BDC9BD975CAC930C45B68D7E820577E61CDA7A968D90742735E60BFB0F4467A7AA0BB12B8B7F6235EB7315E7B59B89C5A4D09"
      },
      {
        "time": 1590431174802,
        "type": "data",
        "data": {
          "property": "44:11:107988:804",
          "seller": 7710138024,
          "buyer": 7710069374,
          "price": 5422285
        },
        "sender": "1e6c708a24317ec06bf92cf4bc100e0532d78a6e6d100e6116fdc09404363b40",
        "hash": "a82b0689315784cb3f51fe01705193ac9b74639b0a29a2ea8da41d3dc7b55e0c",
        "sign": "E953ABE49B765E3CAD47598DE22F1EFA1B2334EA5C9D4B6B6E6920A8C4F703B133122A9F138A2B334BC3789BCDB248418A1443302891582703F21513EBBD8206"
      }
    ],
    "nodes": [
      {
        "ip": "180.240.170.150",
        "port": 1111
      },
      {
        "ip": "190.230.160.140",
        "port": 2222
      }
    ]
  }
}

```

Рисунок 3.12 Пример тела ответа /getNodes

URI: /search

Описание: возвращает результаты поиска.

Параметры: критерии поиска.

На рисунке 3.13 показан пример тела запроса

```

{
  "property": "",
  "anyParty": "",
  "seller": "",
  "buyer": "",
  "minPrice": "3000000",
  "maxPrice": "4000000",
  "startDate": 1590786000000,
  "endDate": 1590958800000
}

```

Рисунок 3.13 Пример тела запроса /search

На рисунке 3.14 показан пример тела ответа.

```

{
  "result": {
    "transactions": [
      {
        "time": 1590844372463,
        "type": "data",
        "data": {
          "property": "40:10:140101:840",
          "seller": "7431259865",
          "buyer": "7135484584",
          "price": "3500000"
        },
        "sender": "475727ef9e6ec139bd0e5b5022afa0eecacabbbcacd586aa0fe839b38b201027",
        "hash": "b26947c2a7fd2890e10c35706d50f899aeeb05c098381c5b1a14648fa26c543f",
        "sign": "1782C9AAE776BB091F42E7D58480583C2774DA604D86BA740E896324E989306D993DCCB6A6315083F78C1105AC06455F4DEA3F59F6743EBB4C40DCEA9F15A40D"
      },
      {
        "time": 1590792284687,
        "type": "data",
        "data": {
          "property": "41:11:141131:844",
          "seller": "7546253102",
          "buyer": "7955456325",
          "price": "3900000"
        },
        "sender": "1e6c708a24317ec06bf92cf4bc100e0532d78a6e6d100e6116fdc09404363b40",
        "hash": "1d39fae3d0ec43064fd4024ca71dda4bb9c1836bd2ce888f1dae734cc24ed4",
        "sign": "81C6848E313ADB5ED359FEA2571588B4C89DF802B498253242EE76880A0AADF8EB93C7E939900650C3977030DFA68CDAFCDC4249F7A8937624D18055E8AD0108"
      }
    ]
  }
}

```

Рисунок 3.14 Пример тела ответа /search

3.5.3. Получение списка мастер-узлов

После составления API для взаимодействия узлов можно перейти непосредственно к реализации сетевого взаимодействия. Разберём, как будет

происходить получение списка доступных мастер узлов в консольном и графическом приложении:

1. При старте приложения менеджер соединений загружает из файла все известные ему мастер-узлы.
2. Менеджер соединений одновременно посылает всем известным мастер-узлам запрос `/getNodes`.
3. Как только от мастер-узла поступает ответ, менеджер соединений заносит этот узел в список доступных узлов вместе с его высотой цепочки. Если через 1 секунду мастер-узел не ответил, он считается недоступным. Помимо этого, в ответе мы получаем новые неподтверждённые транзакции, проверяем и заносим их в пул.
4. Менеджер соединений посылает запрос `/getNodes` на все мастер-узлы, которые содержались в ответе, если эти мастер-узлы ещё не находятся в списке доступных или плохих мастер-узлов, и им ещё не отправлен запрос. В известные мастер-узлы добавляются все мастер-узлы из ответа и сохраняются в файл.
5. Повторяется пункт 3.

То есть такая рекурсия будет повторяться до тех пор, пока все известные для других мастер-узлов мастер-узлы не будут проверены на доступность.

Все запросы отправляются асинхронно. Это значит, что приложение не ждёт ответа от одного узла, чтобы отправить запрос другому узлу. То есть все начальные запросы отправляются одновременно. К тому же при получении ответа от какого-либо узла мы не ждём ответа от других мастер-узлов, а сразу отправляем запросы к мастер-узлам, пришедшим в ответе. Это позволяет существенно снизить время получения доступных мастер-узлов.

На рисунке 3.15 изображён пример получения всех доступных мастер-узлов. На рисунке стрелочками обозначены запросы `/getNodes`, которые приложение отправляет мастер-узлам, цифра рядом со стрелочкой – номер этого мастер узла. Конец стрелки указывает на результат, который мы получили от мастер-узла, а именно все известные ему мастер-узлы. На рисунке

можно увидеть работу описанной выше асинхронности – начальные запросы отправлены одновременно, а отправка одного запроса не ждёт результата другого. Также видно, что приложение не отправляет лишние запросы. Например, оно не отправило запросы на мастер-узлы 4 и 6 при получении ответа от мастер-узла 5, потому что в данный момент мы ожидали ответа от этих мастер-узлов, так как уже отправили на них запросы, когда получили их в ответе от мастер-узла 3. Также запросы не были отправлены на те мастер-узлы, которые нам уже известны.

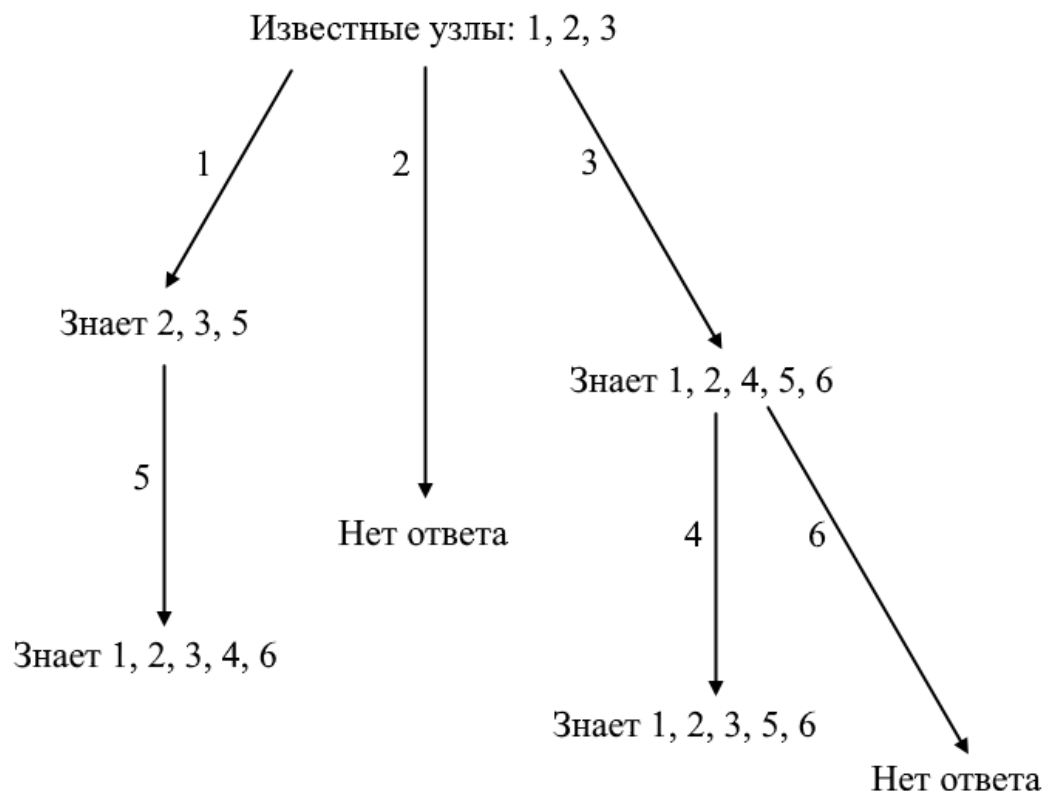


Рисунок 3.15 – пример получения списка доступных мастер-узлов

Данный пример является теоретическим, потому что на практике все мастер-узлы, которым мы отправляем первый запрос, скорее всего уже знали бы друг друга, то есть мы бы получили информацию обо всех мастер-узлах от первого же мастер-узла, соответственно произошло бы всего две итерации цикла.

После того как приложение первый раз соберёт все доступные мастер-узлы, запускается бесконечный цикл, в котором данный процесс повторяется с определённой периодичностью.

3.6. Реализация взаимодействия с цепочкой блоков

При запуске приложения blockchain менеджер производит следующие действия:

1. Загружает из файла цепочку блоков.
2. Получает список валидаторов и отправителей с помощью генезис-блока.
3. Запускает бесконечный цикл обновления.
4. Если мастер-узел является валидатором, запускается цикл валидации.

Разберём подробнее, что представляет из себя каждый цикл.

Цикл обновления с некоторой периодичностью получает новые блоки и новые неподтверждённые транзакции следующим образом:

1. Отправляем на все доступные мастер-узлы запрос /getPendingTransactions.
2. В ответе получаем высоту цепочки блоков мастер-узла. Заносим этот мастер-узел с его высотой во временный список. Помимо этого, в ответе мы получаем новые неподтверждённые транзакции, проверяем и заносим их в пул.
3. Когда получаем все ответы, заменяем старый список доступных мастер-узлов нашим новым получившимся списком.
4. Выбираем из только что обновленного списка доступных мастер-узлов мастер-узел с максимальной высотой.
5. Скачиваем первый недостающий блок с этого мастер-узла, проверяем его, сохраняем в цепочку. Повторяем этот пункт, пока не скачаем все блоки. Если в какой-то момент мастер-узел присылает невалидный блок, то мы возвращаемся к пункту 1. Чтобы не уйти в бесконечный цикл, мы добавляем мастер-узел, который прислал невалидный блок, в список плохих мастер-узлов и удаляем его из списка доступных мастер-узлов. Мастер-узлы, которые находятся в списке плохих мастер-узлов, не попадут в список

доступных мастер-узлов при его обновлении. Соответственно, подключение к ним и загрузка от них новых блоков не будет производиться до перезапуска приложения.

Именно по причине возможности наличия плохих мастер-узлов блоки скачиваются по одному с помощью `/getBlock`, а не все сразу с помощью запроса `/getBlocks`. Если скачивать все блоки сразу и только потом проверять, в случае невалидного блока все последующие блоки придётся скачивать заново. При наличии большого числа плохих мастер-узлов обновление цепочки блоков в таком случае могло бы происходить намного дольше.

Теперь подробнее про цикл валидации. Цикл валидации с определённой периодичностью проверяет имеет ли право этот узел валидации в данный момент подписывать блок. Если да, то начинается процесс подписи – приложение проверяет все транзакции, которые в данный момент находятся в пуле неподтверждённых транзакций, формирует из них блок, подписывает его и добавляет к себе в цепочку.

Через какое-то время с помощью цикла обновления остальные узлы увидят, что высота цепочки блоков у этого валидатора изменилась, и скачают этот блок.

3.7. Создание новой сети

3.7.1. Создание и хранение ключей

Для того чтобы создать сеть, в первую очередь необходимо сгенерировать ключи отправителей и валидаторов.

Для создания ключей было разработано два приложения: веб-приложение и консольная версия.

В целях обеспечения безопасности все закрытые ключи при создании шифруются паролем с помощью симметричного алгоритма шифрования AES. То есть для использования ключей в приложениях необходимо будет каждый раз вводить пароль. Соответственно, даже если

злоумышленник получит доступ к файлу ключа, он не сможет им воспользоваться.

Для создания ключей с помощью веб-приложения (рисунок 3.16) необходимо:

1. Перейти по адресу <https://gval98.github.io>.
2. Ввести пароль два раза.
3. Нажать кнопку «Сгенерировать ключ».
4. Публичный ключ будет отображён в соответствующем поле, закрытый ключ загрузится на компьютер в зашифрованном виде.

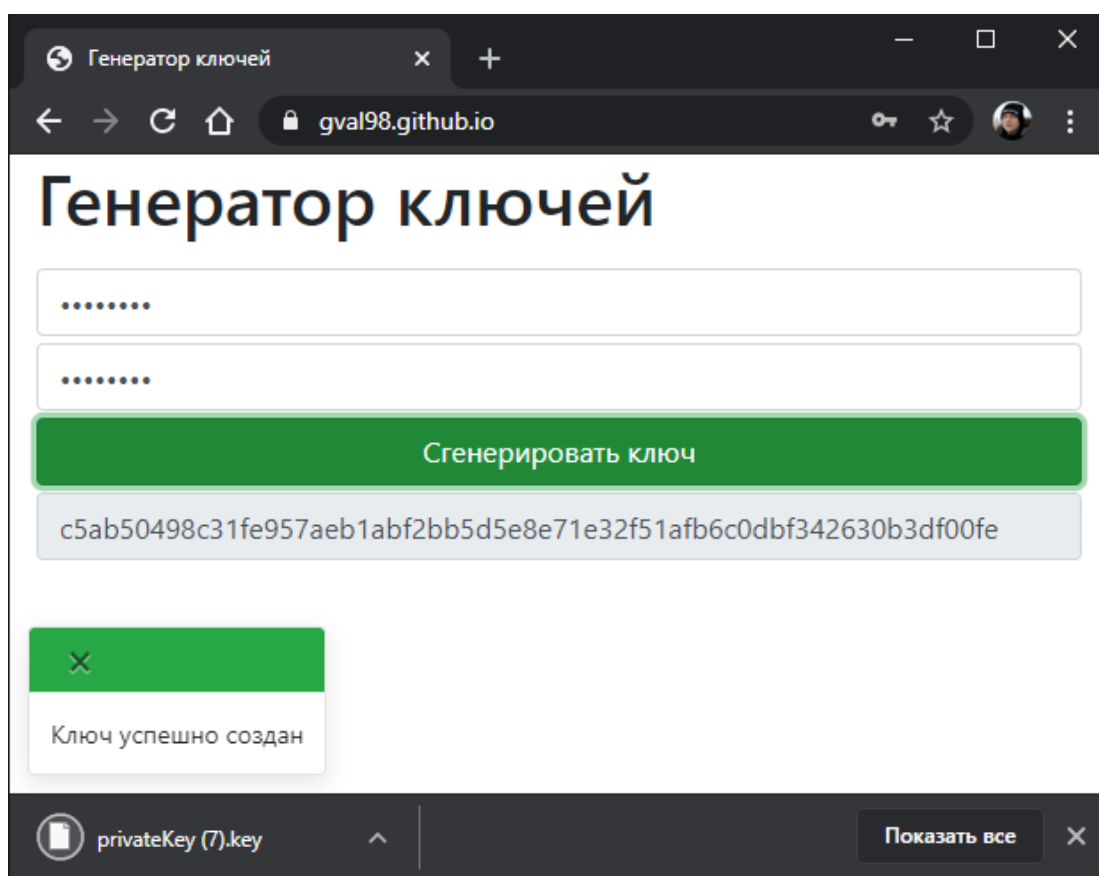


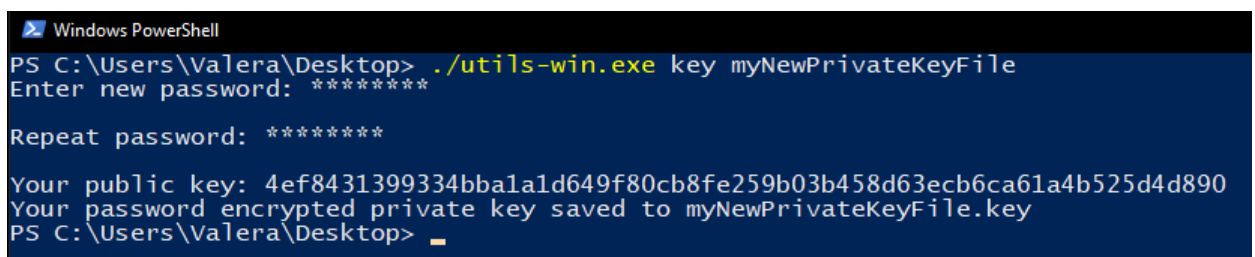
Рисунок 3.16 – веб-версия генератора ключей

Данный способ является безопасным, так как ключи генерируются и зашифровываются непосредственно в браузере, и никакая информация на сервер не передаётся.

Для создания ключей с помощью консольной версии приложения необходимо:

1. Скачать Windows (<http://gval98.mcdir.ru/utills-win.exe>) или Linux версию (<http://gval98.mcdir.ru/utills-linux>).
2. Запустить со следующими параметрами: key имя_файла_нового_приватного_ключа.
3. Ввести пароль два раза.
4. Зашифрованный приватный ключ будет помещён в .key файл с указанным именем, открытый ключ будет выведен на экран.

Пример создания нового ключа в консольной версии на Windows приведён на рисунке 3.17.



```
Windows PowerShell
PS C:\Users\Valera\Desktop> ./utills-win.exe key myNewPrivateKeyFile
Enter new password: *****
Repeat password: *****
Your public key: 4ef8431399334bba1d649f80cb8fe259b03b458d63ecb6ca61a4b525d4d890
Your password encrypted private key saved to myNewPrivateKeyFile.key
PS C:\Users\Valera\Desktop>
```

Рисунок 3.17 Пример создания нового ключа в консольной версии

3.7.2. Создание генезис-блока

Чтобы создать сеть, нужен генезис-блок. Для его автоматического создания используется то же приложение, что и для создания ключей из консоли.

Предположим, что отправители и валидаторы сгенерировали свои пары ключей и отправили открытые ключи человеку, ответственному за создание сети. Для создания генезис-блока этот человек должен:

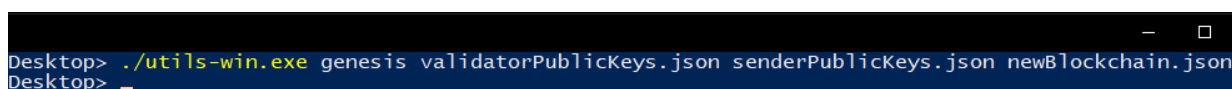
1. Сформировать из ключей валидаторов JSON массив и сохранить в файл.
2. Прodelать тоже самое для ключей отправителей. Пример JSON массива ключей изображён на рисунке 3.18.
3. Запустить приложение со следующими параметрами: genesis файл_ключей_валидаторов файл_ключей_отправителей файл_новой_цепочки_блоков.

4. При успешном выполнении из ключей будет сформирован генезис-блок, из которого будет создана новая цепочка блоков в указанном файле.

Пример создания нового генезис-блока на Windows показан на рисунке 3.19.

```
[  
  "1e6c708a24317ec06bf92cf4bc100e0532d78a6e6d100e6116fdc09404363b40",  
  "a973e616256036a7589ef05f77de0d10c48280f4be2f6323e6148ea290674ddd",  
  "475727ef9e6ec139bd0e5b5022afa0eecacabbbcacd586aa0fe839b38b201027"  
]
```

Рисунок 3.18 Пример JSON массива ключей



```
Desktop> ./utils-win.exe genesis validatorPublicKeys.json senderPublicKeys.json newBlockchain.json  
Desktop> _
```

Рисунок 3.19 Пример создания нового генезис-блока

Для запуска сети необходимо также создать файл с адресами всех валидаторов. Пример приведён на рисунке 3.20.

```
[  
  {  
    "ip": "blockchain-registry.ru",  
    "port": 443  
  },  
  {  
    "ip": "139.28.222.91",  
    "port": 1111  
  },  
  {  
    "ip": "185.244.172.208",  
    "port": 1111  
  }  
]
```

Рисунок 3.20 Пример файла с мастер-узлами

3.8. Установка и запуск консольного приложения

В целях обеспечения удобства установки и запуска приложения оно было запаковано в исполняемые файлы с помощью утилиты pkg.

Для установки приложения необходимо скачать и распаковать архив cli-win.zip или cli-linux.tar. Скачать их можно по ссылкам

<http://gval98.mcdir.ru/cli-win.zip> и <http://gval98.mcdir.ru/cli-linux.tar>. В данных архивах находятся: список узлов и генезис-блок для уже запущенной сети, исполняемый файл.

Консольное приложение должно запускаться со следующими параметрами: IP-адрес запускаемого мастер-узла, порт запускаемого мастер-узла, путь к файлу с мастер-узлами, путь к файлу с цепочкой блоков, путь к файлу с ключом валидатора, пароль от ключа валидатора, путь к файлу SSL сертификата, ключ от SSL сертификата. Пример параметров запуска мастер-узла валидации: 190.240.10.150 443 nodes.json blockchain.json validatorPrivateKey.key Valera16 sslCert.cert sslKey.key. Если приложение запускается в качестве мастер-узла без валидации, и вы хотите на лету сгенерировать SSL сертификаты, в соответствующих параметрах необходимо поставить 0. Пример: 190.240.10.150 1111 nodes.json blockchain.json 0 0 0 0

Пример успешного запуска узла валидации с использованием файла сертификатов показан на рисунке 3.21.

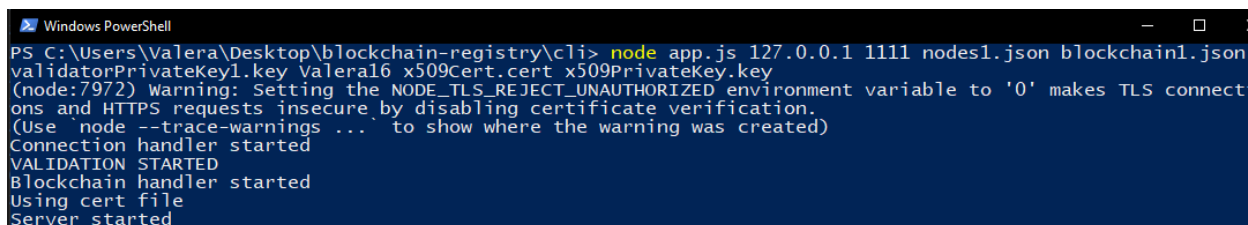


Рисунок 3.21 Запуск консольного приложения

3.9. Создание приложения с графическим интерфейсом

3.9.1. Проектирование и разработка

Для создания графического интерфейса использовался фреймворк Electron. Данный фреймворк позволяет разрабатывать кроссплатформенные приложения с графической оболочкой для Windows, Linux и MacOS.

Для рендера Electron использует веб-технологии. По сути, он использует браузерный движок для рендера веб-страниц. То есть для разработки графической части приложения используется язык гипертекстовой разметки

HTML вместе с CSS и JavaScript. Использование HTML и CSS является большим преимуществом, так как эти технологии предоставляют очень гибкие инструменты для создания пользовательских элементов, к тому же имеется множество готовых фреймворков для удобной и эффективной работы с ними. JavaScript тоже является преимуществом, так как используется во всём проекте.

Приложение с графическим интерфейсом будет написано на основе консольного приложения. То есть графическая веб-часть приложения будет общаться с консольной частью приложения для получения данных и отрисовывать их в удобном виде. Для реализации такого функционала в Electron существует событийная модель. Например, при начальной загрузке веб-страницы в ней вызывается событие “windowReady”. В консольной части приложения срабатывает обработчик этого события, который вызывает событие “newBlock”, в котором указывает данные последних пяти блоков. На веб-странице срабатывает обработчик этого события, он получает данные о блоках в нём и отрисовывает эту информацию. События могут также изначально вызываться и в консольной части приложения, например при получении нового блока или транзакции.

Структура графического приложения идентична структуре консольного за исключением того, что вместо папки “web” у нас теперь папка “gui”. Файлы, которые она содержит, и их описание приведено в таблице 3.7.

Таблица 3.7 – содержание папки “gui”

Файл или папка	Описание
bootstrap/	Папка, хранящая файлы фреймворка Bootstrap
font/	Папка, хранящая иконки
electron-app.js	Инициализирует запуск графического интерфейса
index.html	Файл HTML разметки
renderer.js	Подключается в HTML файле. Отвечает за интерактивность веб-страницы, обработку событий, присылаемых консольной частью, и отрисовку полученных данных
style.css	Таблица стилей

Инициализация приложения отличается от консольного и происходит следующим образом:

1. Запускается точка хода в приложение, в которой подключаются все необходимые для старта модули.
2. Приложение инициализирует графический интерфейс.
3. Приложение ждёт, когда графический интерфейс будет запущен.
4. Приложение запускает менеджер подключений.
5. Приложение ждёт, когда менеджер подключений подключится к мастер-узлам.
6. После этого приложение запускает blockchain менеджер.
7. Приложение ждёт, когда blockchain менеджер загрузит из файла уже имеющуюся цепочку блоков.

В этом случае, как мы видим, по умолчанию функционал мастер-узла отключен, и HTTP-сервер не запускается автоматически при старте приложения.

Приступим к разработке интерфейса. В первую очередь нам необходимо сверстать веб-страницу. Для этих целей был использован CSS фреймворк Bootstrap. Он предоставляет HTML и CSS инструменты для быстрой и удобной вёрстки веб-страниц. Он включает в себя шаблоны кнопок, списков, карточек, форм, готовые элементы для позиционирования блоков. Для удобной работы с элементами веб-страницы также использовалась библиотека jQuery.

3.9.2. Графический интерфейс

Главный экран приложения изображён на рисунке 3.22. Рассмотрим подробнее основные части интерфейса.

В левой части находится блок с новыми транзакциями (Рисунок 3.23). Здесь в виде карточек отображаются все транзакции, которые в данный момент находятся в пуле неподтверждённых транзакций. Этот список обновляется в реальном времени. Снизу карточки отображается публичный ключ отправителя транзакции. В случае новых транзакций, сверху карточки

отображается время, которое указал отправитель, его нельзя считать 100% достоверным.

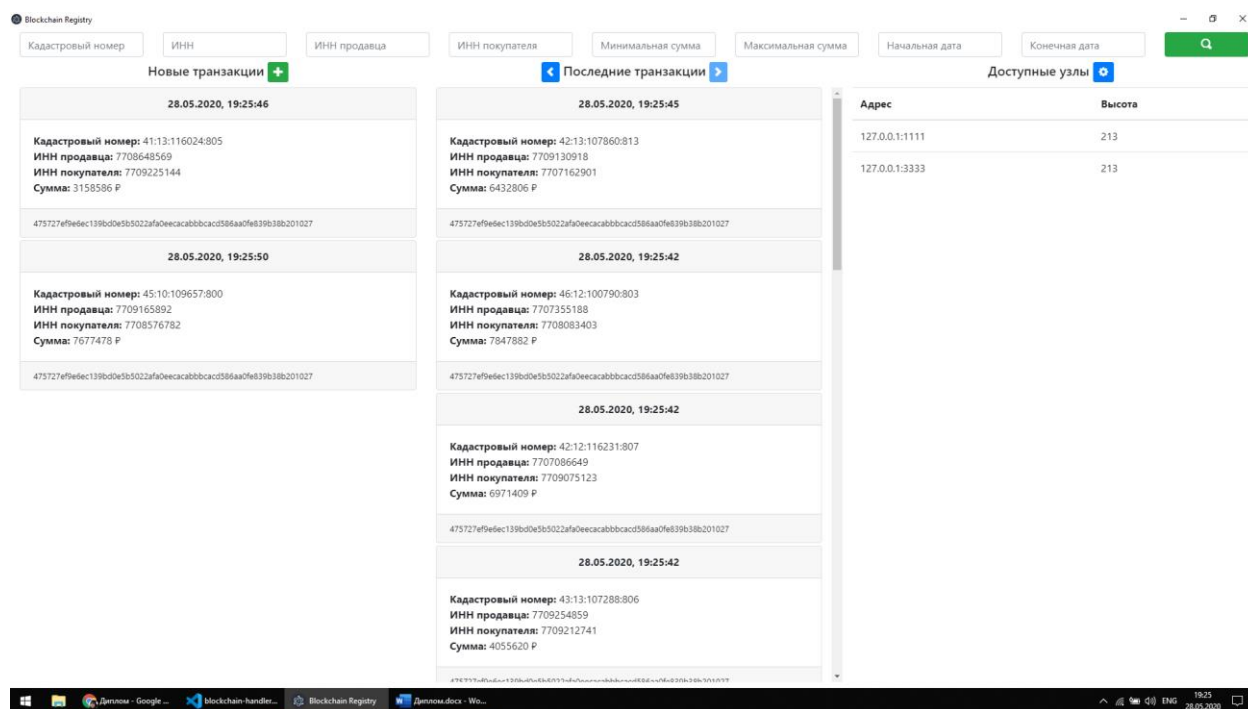


Рисунок 3.22 Главный экран приложения

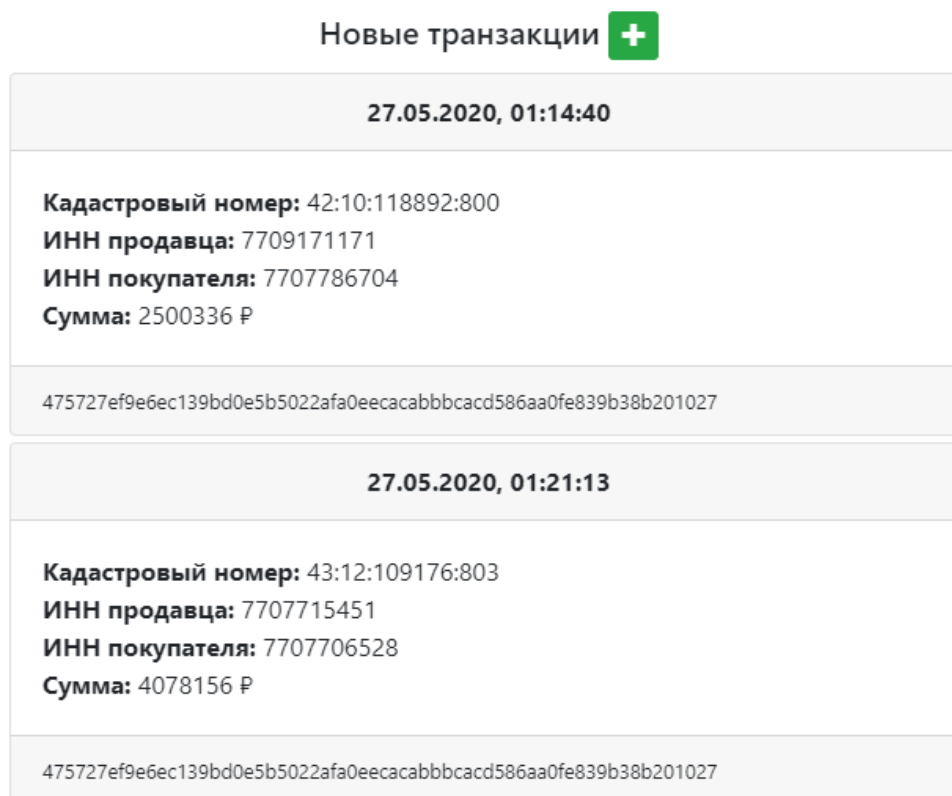


Рисунок 3.23 Новые транзакции

При нажатии на зелёную кнопку с плюсом произойдёт открытие диалогового окна для добавления новой транзакции (Рисунок 3.24). Здесь необходимо выбрать ключ отправителя. При нажатии на это поле откроется окно выбора файла (Рисунок 3.25), в котором необходимо выбрать нужный .key файл. Также нужно будет ввести пароль от ключа и все необходимые данные для самой транзакции.

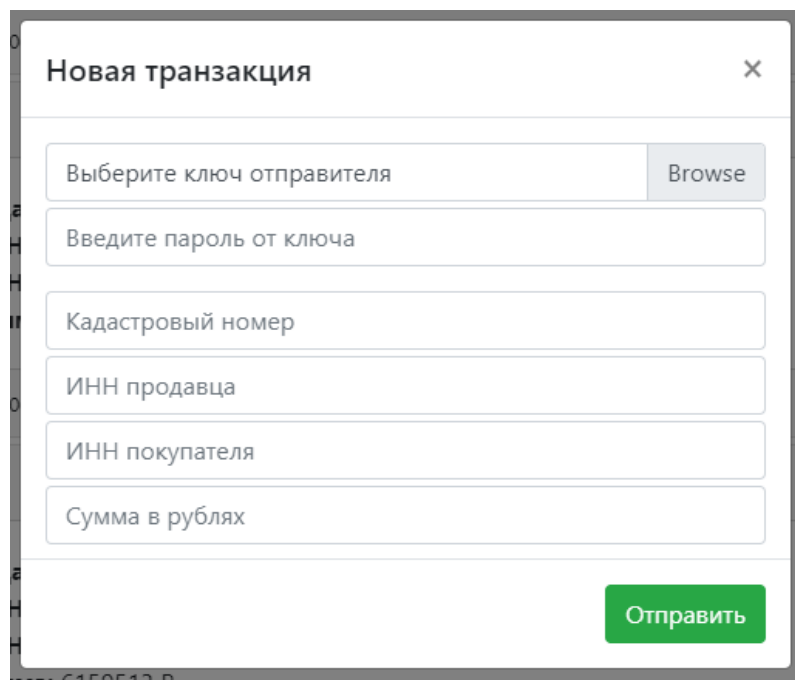


Рисунок 3.24 Диалоговое окно добавления транзакции

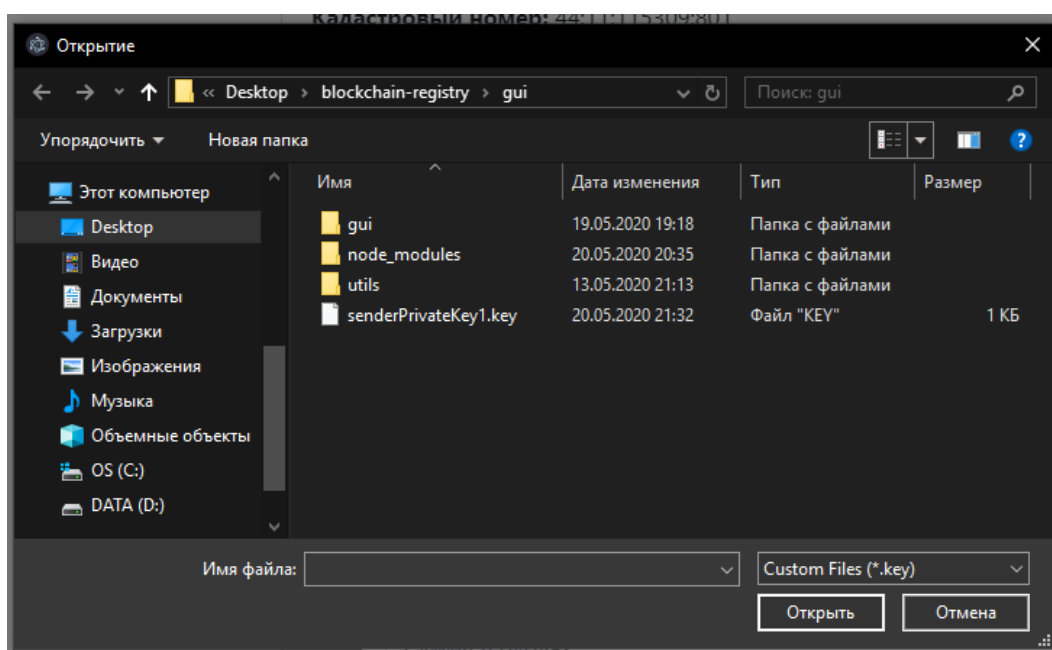


Рисунок 3.25 Диалоговое окно выбора ключа

Если все данные введены верно, то при нажатии на кнопку «Отправить» диалоговое окно закроется, мы увидим уведомление о том, что транзакция успешно отправлена. Когда мастер-узлы обработают её, она тут же появится в списке новых транзакций (Рисунок 3.26). Когда валидатор подпишет блок с этой транзакцией, она сразу исчезнет из списка новых транзакций и появится в списке последних транзакций.

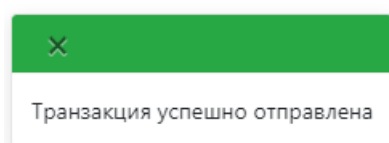
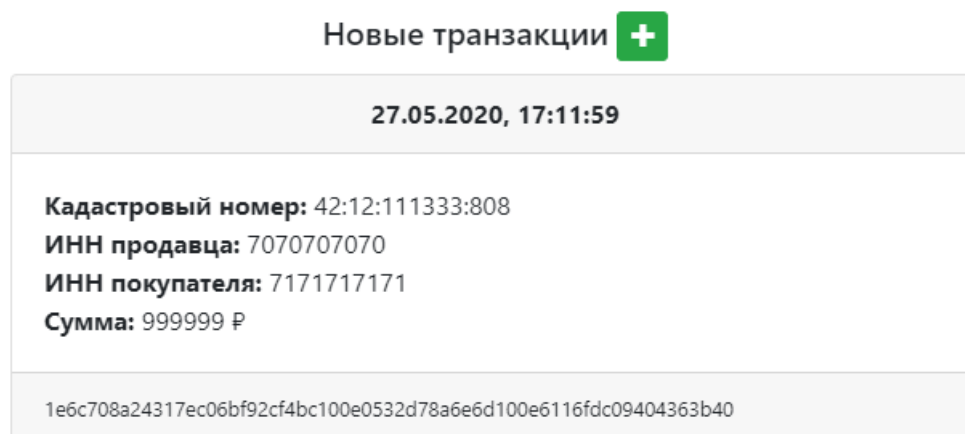


Рисунок 3.26 Успешное добавление транзакции

Если же данные неверные, диалоговое окно не закроется, а мы увидим сообщение об ошибке (Рисунок 3.27). Здесь также видно, что имя файла ключа после его выбора отображается в соответствующее поле, а введенный пароль отображается только в зашифрованном виде.

В правой части интерфейса отображаются все доступные в данный момент мастер-узлы и высота их цепочки блоков (Рисунок 3.28). Этот список

обновляется в реальном времени. При нажатии на синюю кнопку с шестерёнкой откроется диалоговое окно для запуска мастер-узла (Рисунок 3.29). В этом окне необходимо ввести свой внешний IP-адрес и порт, на котором будет запущен мастер узел. После нажатия на кнопку «Запустить», при успешном запуске HTTP-сервера в углу экрана будет отображено соответствующее сообщение, а диалоговое окно будет закрыто.

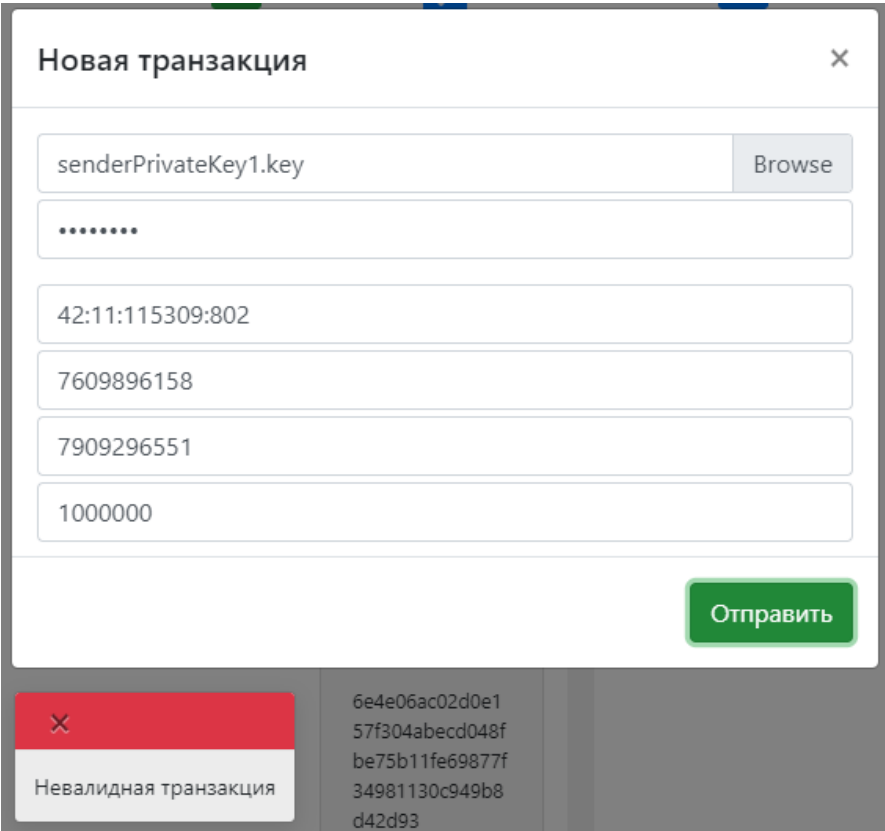



Рисунок 3.27 Неудачное добавление транзакции

Доступные узлы 

Адрес	Высота
139.28.222.91:1111	16
blockchain-registry.ru:443	16
185.244.172.208:1111	16

Рисунок 3.28 Доступные узлы

Рисунок 3.29 Диалог запуска мастер-узла

В центральной части приложения показываются транзакции из последних пяти блоков (Рисунок 3.30). Этот список обновляется в реальном времени.

<div> <div></div> <div>Последние транзакции</div> <div></div> </div>	
27.05.2020, 01:13:14	
Кадастровый номер: 42:13:104448:809 ИНН продавца: 7707755725 ИНН покупателя: 7708015970 Сумма: 8597445 ₽	
475727ef9e6ec139bd0e5b5022afa0eecacabbcbacd586aa0fe839b38b201027	
20.05.2020, 21:32:01	
Кадастровый номер: 45:13:111431:812 ИНН продавца: 7708644190 ИНН покупателя: 7710074381 Сумма: 5772177 ₽	
1e6c708a24317ec06bf92cf4bc100e0532d78a6e6d100e6116fdc09404363b40	
20.05.2020, 21:32:01	
Кадастровый номер: 41:11:100295:807 ИНН продавца: 7708980711 ИНН покупателя: 7707803960 Сумма: 5402688 ₽	
a973e616256036a7589ef05f77de0d10c48280f4be2f6323e6148ea290674ddd	
20.05.2020, 21:32:01	
Кадастровый номер: 45:14:106722:807 ИНН продавца: 7708108456 ИНН покупателя: 7708065928 Сумма: 5754035 ₽	
475737af0c60c120bd0a5b5022afa0eecacabbcbacd586aa0fe839b38b201027	

Рисунок 3.30 Последние транзакции

Как мы видим, здесь присутствует возможность переключения страниц для перехода к более ранним блокам. Если пользователь перешёл на какую-либо страницу, которая не является основной, автоматическое обновление этой части интерфейса уже не происходит. При возврате пользователя на основную страницу новые транзакции будут отображены и продолжают обновляться в реальном времени. Это также отображается в изменении названия центральной части (Рисунок 3.31)

Транзакции	
19.05.2020, 15:04:58	
Кадастровый номер: 44:11:115309:801 ИНН продавца: 7709896159 ИНН покупателя: 7708712708 Сумма: 7016082 Р	
6e4e06ac02d0e157f304abecd048fbe75b11fe69877f34981130c949b8d42d93	
19.05.2020, 15:04:28	
Кадастровый номер: 47:12:104198:803 ИНН продавца: 7707589138 ИНН покупателя: 7708956767 Сумма: 3755406 Р	
6e4e06ac02d0e157f304abecd048fbe75b11fe69877f34981130c949b8d42d93	
16.05.2020, 23:45:15	
Кадастровый номер: 44:12:114082:811 ИНН продавца: 7708957853 ИНН покупателя: 7708726340 Сумма: 6159513 Р	
6e4e06ac02d0e157f304abecd048fbe75b11fe69877f34981130c949b8d42d93	
16.05.2020, 23:45:00	
Кадастровый номер: 40:12:110389:803 ИНН продавца: 7707596953 ИНН покупателя: 7709956878 Сумма: 5102713 Р	

Рисунок 3.31 Переключение страниц

В верхней части приложения расположен поиск (Рисунок 3.32).

Кадастровый номер ИНН ИНН продавца ИНН покупателя Минимальная сумма Максимальная сумма Начальная дата Конечная дата **Q**

Рисунок 3.32 Панель поиска

С помощью поиска мы можем фильтровать транзакции по заданным критериям. Результаты поиска отображаются в центральной части экрана. При этом автоматическое обновление этой части интерфейса уже не происходит. При сбросе фильтров новые транзакции будут отображены и продолжат обновляться в реальном времени. Также будет изменено название центральной части интерфейса.

Например, мы хотим узнать, какие сделки 27 мая 2020 года на сумму больше 5 000 000 и меньше 9 000 000 совершил человек, ИНН которого 7707755725. Для этого заносим в поиск следующие данные:

- ИНН – 7707755725
- Минимальная сумма – 5000000
- Максимальная сумма – 9000000
- Начальная дата – 27.05.2020
- Конечная дата – 27.05.2020

Полученные результаты показаны на рисунке 3.33.

< Результаты ✕ >

27.05.2020, 18:01:29
Кадастровый номер: 41:14:131123:838 ИНН продавца: 7070707070 ИНН покупателя: 7707755725 Сумма: 6000000 Р
1e6c708a24317ec06bf92cf4bc100e0532d78a6e6d100e6116fdc09404363b40
27.05.2020, 01:13:14
Кадастровый номер: 42:13:104448:809 ИНН продавца: 7707755725 ИНН покупателя: 7708015970 Сумма: 8597445 Р
475727ef9e6ec139bd0e5b5022afa0eecacabbbccacd586aa0fe839b38b201027

Рисунок 3.33 Результаты поиска

3.9.3. Установка

Для обеспечения удобной установки приложения была использована утилита electron-builder. Она позволяет упаковывать приложения и создавать установщики для Windows, MacOS и Linux.

В качестве установщика для Windows в electron-builder был выбран NSIS – система создания установочных программ для Windows с открытым исходным кодом.

Теперь для того, чтобы установить приложение достаточно запустить установочный файл, указанный на рисунке 3.34. Скачать его можно по ссылке <http://gval98.mcdir.ru/Blockchain%20Registry%20Setup%201.0.0.exe>

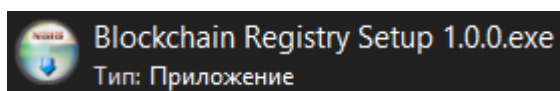


Рисунок 3.34 Установочный файл для Windows

При запуске откроется окно установки (рисунок 3.35), в котором нужно будет выбрать директорию для установки программы.

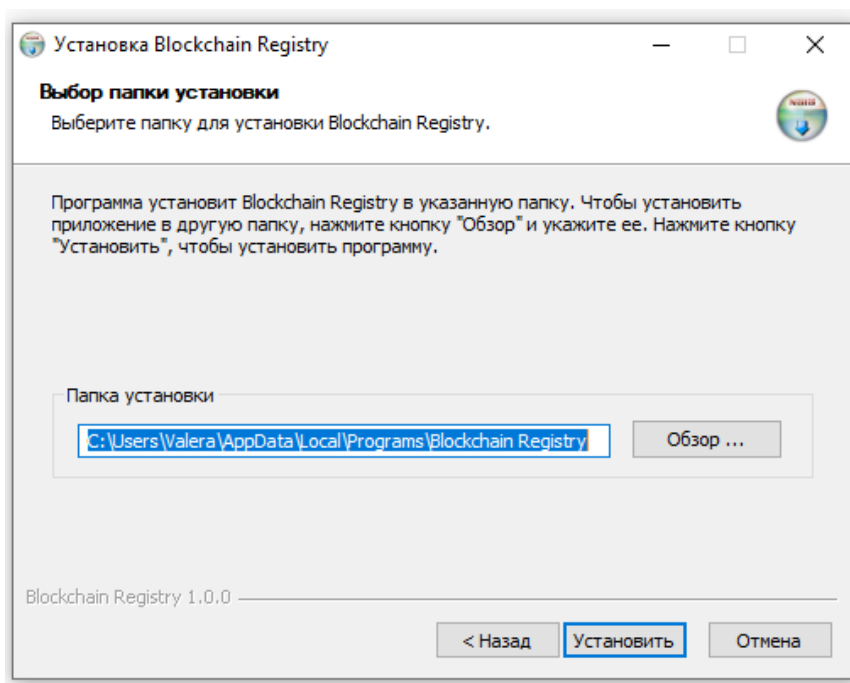


Рисунок 3.35 Установка приложения

После окончания установки будет создан ярлык на рабочем столе (рисунок 3.36), с помощью которого можно будет удобно запускать приложение.



Рисунок 3.36 Ярлык Windows приложения

Для запуска Linux версии необходимо запустить исполняемый файл, указанный на рисунке 3.37. Скачать его можно по ссылке <http://gval98.mcdir.ru/Blockchain%20Registry-1.0.0.AppImage>



Рисунок 3.37 Исполняемый файл для Linux

3.10. Разработка веб-приложения

Как говорилось в пункте 3.2, основной особенностью веб-версии является то, что она не хранит всю цепочку блоков, а только запрашивает нужные у доверенного мастер-узла. По сути, веб-приложение, в отличие от консольного и графического, является тонким клиентом. Основным его преимуществом является то, что оно доступно на любом устройстве с веб-браузером и не занимает место на диске. Веб-приложение не проверяет блоки и транзакции, поэтому мастер-узел должен являться доверенным. Например, чтобы не скачивать всю цепочку на несколько компьютеров в локальной сети, можно запустить один мастер-узел и обращаться к нему с помощью веб-приложения.

Доступ к веб-интерфейсу можно получить по следующему адресу:
https://адрес_мастер_узла:порт/web

По сути, при обращении к этому адресу мастер-узел только предоставляет веб-интерфейс, который мог бы быть доступен и локально, и с другого домена, если бы не современные ограничения браузеров, из-за которых адрес веб-интерфейса должен совпадать с адресом мастер-узла. К тому же мастер-узел должен использовать доверенные, а не самоподписанные SSL сертификаты, иначе браузер выдаст предупреждение. Так как в графическом приложении имеется возможность запускать мастер-узел только с самоподписанными сертификатами, оно не предоставляет доступ к веб-интерфейсу.

Логика работы веб-приложения следующая:

1. При запуске приложение запрашивает у мастер-узла транзакции из последних пяти блоков.
2. С определённым интервалом запрашиваются новые неподтверждённые транзакции и высота мастер-узла.
3. Если высота больше текущей, приложение запрашивает транзакции из последних пяти блоков.

К тому же присутствуют все функции графического приложения, такие как:

1. Подпись и добавление транзакции. Это является безопасным, так как подпись происходит прямо в браузере.
2. Поиск транзакций. Происходит с помощью API.
3. Переключение страниц для отображения более ранних транзакций. Происходит с помощью API.

Так как приложение должно поддерживать мобильные браузеры, интерфейс создавался адаптивным. То есть он подстраивается под размеры экрана. Интерфейс веб-приложения на смартфоне показан на рисунке 3.38.

16:37 📶 📶 95

🏠 blockchain-registry.ru/web 1 ⋮

Кадастровый номер

ИНН

ИНН продавца

ИНН покупателя

Минимальная сумма

Максимальная сумма

Начальная дата

Конечная дата

🔍

◀ Последние транзакции ▶

30.05.2020, 16:13:30

Кадастровый номер: 40:101140101:841
ИНН продавца: 7424559659
ИНН покупателя: 7143584666
Сумма: 4500000 ₽

475727ef9e6ec139bd0e5b5022afa0eecacabbbcacd586
aa0fe839b38b201027

Рисунок 3.38 Веб-приложение, запущенное в браузере смартфона

В мобильной версии приложение пункт «Новые транзакции» находится под пунктом «Последние транзакции». Соответственно, переключение между

ними осуществляется с помощью прокрутки. Поля поиска теперь занимают всю ширину экрана.

Так как графическое приложение уже разрабатывалось с применением HTML, CSS и JS, интерфейс для больших экранов остался практически таким же (Рисунок 3.39).

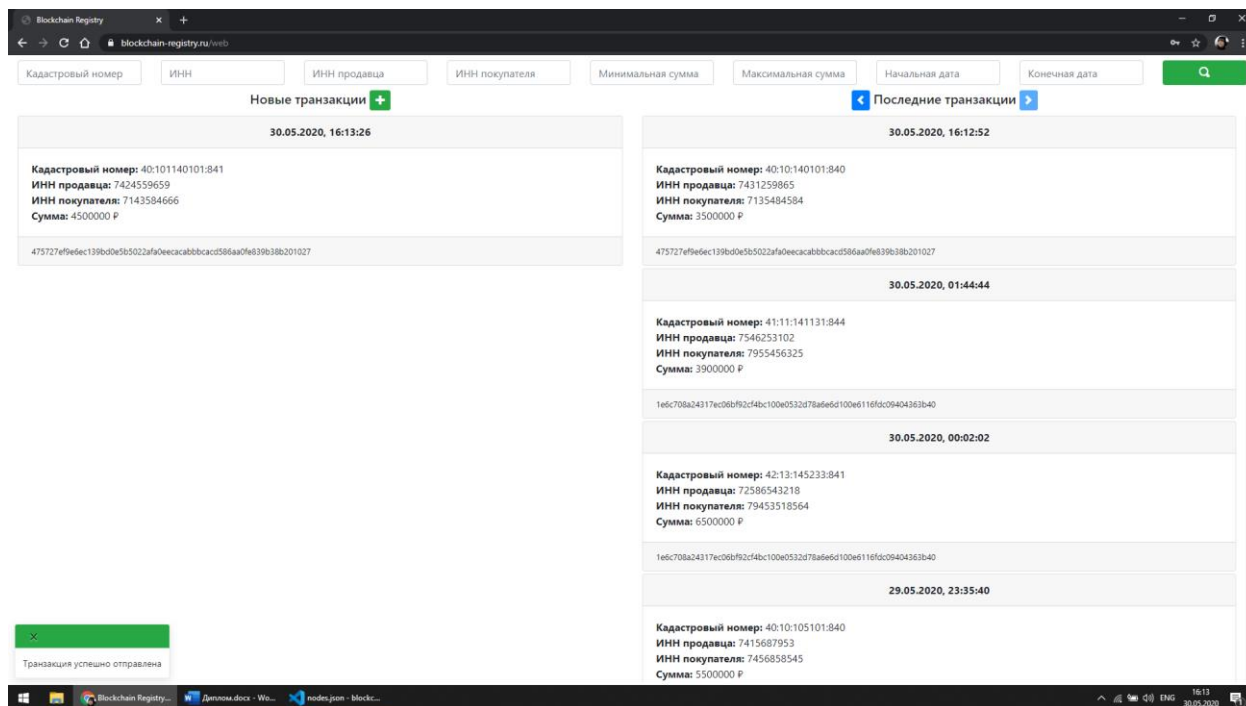


Рисунок 3.39 Веб-приложение

Как мы видим, в правой части интерфейса отсутствует пункт «Доступные узлы», потому что приложение обращается только к одному мастер-узлу. Теперь пункты «Новые транзакции» и «Последние транзакции» занимают ровно по половине экрана.

Веб-приложение для уже запущенной сети доступно по адресу <https://blockchain-registry.ru/web>

ЗАКЛЮЧЕНИЕ

В результате выполнения выпускной квалификационной работы были проанализированы существующие решения для организации реестров хранения данных и разработано кроссплатформенное программное обеспечение для организации публичного децентрализованного реестра и графический интерфейс для удобной работы с ним.

В ходе выполнения работы были решены следующие задачи:

1. Рассмотрены преимущества и недостатки существующих в данный момент электронных реестров.
2. Рассмотрены существующие решения для децентрализованного обмена данными, в том числе с использованием технологии blockchain. Обоснована актуальность работы.
3. Произведён разбор технологии blockchain и обоснование её использования для создания публичного децентрализованного реестра.
4. Рассмотрены такие понятия, как пиринговые сети, хеш-функции, эллиптическая криптография, алгоритм консенсуса.
5. Произведён сравнительный анализ технологий, используемых в blockchain проектах и децентрализованных сетях. В результате были выбраны наиболее подходящие для организации распределённого реестра технологии и определена его структура.
6. Определены требования к разрабатываемому программному обеспечению и выбраны наиболее соответствующие им средства разработки.
7. В целях обеспечения удобства создания новой сети разработано консольное приложение на Windows и Linux, позволяющее генерировать новые ключи и генезис-блоки, а также графическая веб-версия приложения для создания ключей.
8. Разработано консольное приложение на Windows и Linux для мастер-узлов и валидаторов, с помощью которых обеспечивается надёжность сети, происходит получение и распространение новой информации.

9. Разработано приложение с графическим интерфейсом для Windows и Linux на русском языке, позволяющее пользователю удобно просматривать все имеющиеся на данный момент данные, в реальном времени получать новую информацию, добавлять новые записи, производить поиск данных по заданным критериям.

10. Разработано веб-приложение, позволяющее производить все действия, доступные в графическом приложении, при этом не занимающее место на диске и доступное на любом устройстве с веб-браузером.

11. Развёрнута сеть, состоящая из трёх валидаторов, подключиться к которой можно, используя разработанные приложения.

Исходя из результатов проделанной работы можно сделать вывод, что разработанные в ходе выполнения выпускной квалификационной работы приложения могут быть успешно применены для создания публичных реестров на базе технологии Blockchain с целью их децентрализации, увеличения надежности и прозрачности.

СПИСОК ЛИТЕРАТУРЫ

1. Erdely R., Kerle T., Levine B., Liberatore M., Shields C. Forensic Investigation of Peer-to-Peer File Sharing Network // Digital Investigation. 2010. Iss. 7. P. 95–103.
2. Benet J. IPFS - Content Addressed, Versioned, P2P File System // arXiv: Networking and Internet Architecture. 2014. 11 p.
3. Yaga D., Mell P., Roby N., Scarfone K. Blockchain Technology Overview // National Institute of Standards and Technology Interagency Report. 2018. 68 p.
4. Storj: A Decentralized Cloud Storage Network Framework [Electronic resource] // Storj Labs, Inc. 2018. URL: <https://storj.io/storj.pdf> (Access date: 04.06.2020).
5. DApp Statistics [Electronic resource] // State of the DApps. 2020. URL: <https://www.stateofthedapps.com/stats> (Access date: 04.06.2020).
6. Hu Y., Liyanage M., Manzoor A., Thilakarathna K., Jourjon G., Seneviratne A. Blockchain-based Smart Contracts – Applications and Challenges // arXiv: Computers and Society. 2019. 26 p.
7. Stevens M. Fast Collision Attack on MD5 // Cryptology ePrint Archive. 2006. 13 p.
8. Stevens M., Bursztein E., Karpman P., Albertini A., Markov Y. The first collision for full SHA-1 // Advances in Cryptology – CRYPTO 2017. 2017. P. 570–596.
9. Wang L., Shen X., Li J., Shao J., Yang Y. Cryptographic primitives in blockchains // Journal of Network and Computer Applications. 2019. Iss. 127. P. 43–58.
10. Sinha R., Srivastava H.K., Gupta S. Performance Based Comparison Study of RSA and Elliptic Curve Cryptography // International Journal of Scientific & Engineering Research. 2013. Iss. 4. P. 720–725.

11. Bernstein D.J., Lange T. SafeCurves: choosing safe curves for elliptic-curve cryptography [Electronic resource]. URL: <https://safecurves.cr.yp.to/> (Access date: 24.05.2020).
12. Josefsson S., Liusvaara I. Edwards-Curve Digital Signature Algorithm (EdDSA) // RFC 8032. 2017. 60p.
13. Buford J.F., Yu H. Peer-to-Peer Networking and Applications: Synopsis and Research Directions // Handbook of Peer-to-Peer Networking. 2010. 43p.
14. Xiao Y., Zhang N., Lou W. A Survey of Distributed Consensus Protocols for Blockchain Networks // IEEE Communications Surveys & Tutorials, vol. 22, no. 2. 2020. P. 1432-14