Санкт-Петербургский государственный университет

Системное программирование
Математическое обеспечение и администрирование информационных систем

Смирнов Денис Павлович

Разработка криптовалютного шлюза

Бакалаврская работа

Научный руководитель:
доц. кафедры СП, к. ф.-м. н. К. Ю. Романовский

Консультант:
ст. преп. кафедры СП, Я. А. Кириленко

Санкт-Петербург
2020

Saint Petersburg state university

System Programming
Software and Administration of Information Systems

Denis Smirnov

Cryptocurrency gateway development

Bachelor's Thesis

Scientific supervisor:
Associate Professor, Ph.D. Konstantin Romanovsky

Adviser:
Senior Lecturer, Iakov Kirilenko

Saint Petersburg
2020

Оглавление

Введение

За последние двадцать лет интернет торговля стала неотъемлемой частью экономики всех развитых стран мира. Согласно отчёту [2] сооснователя Bond Capital Мэри Микер за 2019 год доля электронной торговли в процентах от общего объема розничных продаж в США увеличилась с 1% в 2000 году до 16% в 2019. Активное развитие этой области очень быстро привело к возникновению специфических аппаратно-программных комплексов, так называемых платёжных шлюзов, которые предназначены для облегчения и автоматизации электронных платежей. Самым известным среди них является PayPal, и один только он насчитывает 180 миллионов активных пользователей [12].

С появлением в 2008 году Биткойна, в корне изменилось представление о том, как может выглядеть финансовая система. Биткойн — это исторически первая в мире криптовалюта, новый вид цифрового актива, которым можно без посредников совершать мгновенные платежи любому участнику биткойн-сети. Биткойн не использует никакого центрального органа управления: обработка транзакций и эмиссия денег осуществляются коллективно всей сетью. В основе его работы лежит особый вид децентрализованной базы данных, называемой блокчейном. Она гарантирует сохранность информации и определяет строгие правила, позволяющие добавлять туда новые записи с сохранением консистентности состояния всех узлов сети. Неизвестно, кто именно занимался созданием этой системы, но все механизмы её работы были опубликованы в статье [10], а исходный код полностью открыт. Модель, предлагаемая биткойном, оказалась настолько впечатляющей и эффективной, что сообщества разработчиков по всему миру начали создавать свои собственные криптовалюты, и на данный момент их насчитывается уже несколько тысяч [3].

Многие считают, что децентрализованные сети по своей сути совершеннее укоренившихся банковских систем и в скором времени полностью их заменят. Достоверно утверждать этого, разумеется, нельзя, централизованные системы имеют ряд своих преимуществ (напри-

мер, они более производительные). Но уже сейчас существует достаточный спрос на программное обеспечение, которое, подобно традиционным платежным шлюзам, позволяет легко и безболезненно интегрировать криптовалютную платёжную систему в различного рода интернет-магазины, и есть все основания считать, что в будущем этот спрос будет только увеличиваться. Соответственно, возникает необходимость в создании ПО, которое, предоставляя единый интерфейс, позволяет отслеживать транзакции и производить платежи самыми разными криптовалютами.

1. Постановка задачи

Целью данной работы является создание шлюза, предоставляющего единый интерфейс для взаимодействия с разными криптовалютами и позволяющего отслеживать транзакции и инициировать новые платежи.

Для достижения цели были поставлены следующие задачи.

1. Провести анализ существующих решений.
2. Спроектировать интерфейс.
3. Реализовать интерфейс для криптовалют Monero и Bitcoin (выбор валют обусловлен необходимостью их поддержки конкретным внешним сервисом, для взаимодействия с которым разрабатывается шлюз).
4. Обеспечить возможность восстановления состояния системы в случае аварийного завершения работы и частичной потери данных.
5. Реализовать шлюз, способный работать на двух платформах: Linux и Windows.
6. Реализовать тестовое окружение и протестировать на нём шлюз.

2. Технические требования

Важно отметить несколько моментов. Криптовалютный шлюз, представленный в данной ВКР, разрабатывался специально для функционирования с некоторым вполне конкретным сервисом, у которого есть ряд важных характеристик. Это высоконагруженное приложение, которому критически важна возможность самостоятельно балансировать свою нагрузку. Его старались проектировать так, чтобы минимизировать все возможные накладные расходы (создание новых потоков, исключения в коде, сборку мусора). Каждая коммуникация сервиса с внешним миром должна проходить за близкое к константному время, поэтому никакие долгие сетевые запросы он делать не может. Поскольку криптографические вычисления в некоторых валютах могут занимать целую вечность по меркам современных компьютеров (вплоть до секунды на генерацию транзакции в Monero!) из этого следует, что интерфейс шлюза должен быть асинхронным. Также у сервиса есть своё собственное персистентное хранилище данных, поэтому отдельная БД шлюзу не нужна. Наконец, большая часть программистов, занимающихся разработкой и сопровождением проекта, программируют, в основном, на C++, и имеют много опыта связанного именно с ним.

Исходя из всего вышеперечисленного, основные технические требования к шлюзу изначально формулировались следующим образом.

- Асинхронный интерфейс.
- Поддержка трансляции специфичных для каждой криптовалюты данных в единый унифицированный формат, понимаемый сервисом, и обратно. Данные пользователей хранит сервис, данные блокчейна — криптовалютные узлы и кошельки. Шлюз отвечает за генерацию и отслеживание адресов сервиса, сообщение о новых транзакциях на его адреса, вывод денег на указанный адрес назначения и валидацию адресов в каждом конкретном блокчейне.
- Выбор C++в качестве языка разработки.

По ходу работы, однако, добавились несколько новых нюансов. Все изменения шлюза старались как можно быстрее интегрировать с сервисом, который с первых недель был запущен на тестовых блокчейнах, поэтому для обновлений шлюза его приходилось систематически останавливать, пересобирать и запускать заново, что было неудобно. Кроме того, быстро стало понятно, что было бы полезно иметь возможность запускать сервис и шлюз на физически разных серверах, возможно даже в нескольких экземплярах. В довершение, поскольку любой сервис, работающий с деньгами, должен стремиться сводить к минимуму время работы в нештатном режиме, возникло желание максимально упростить логику его остановки в случае форс-мажора. Всё это привело к формулированию ещё одного требования.

Сам шлюз должен быть реализован как отдельное приложение, которое предоставляет свой интерфейс для взаимодействия и общается с внешним миром через него.

3. Обзор предметной области

В этом разделе содержится информация об уже существующих криптовалютных шлюзах, краткое определение ключевых понятий из предметной области, а также описание алгоритма, необходимого для понимания важной детали реализации шлюза Monero.

3.1. Существующие решения

Криптовалюты появились уже достаточно давно для того, чтобы успела сформироваться потребность в платёжных криптошлюзах, подобных шлюзам для фиатных денег. На данный момент на рынке существует ряд компаний, предлагающих предпринимателям криптовалютные шлюзы в качестве отдельной услуги. Известными и популярными продуктами, поддерживающими хотя бы несколько десятков криптовалют, являются, например, решения от CoinPayments, Coingate и Paycoiner. К сожалению, у этих сервисов существует ряд недостатков, затрудняющих их использование.

Во-первых, они кастодиальные, то есть все ключи от активов пользователей находятся у третьих лиц. Это означает, что для использования сервиса необходимо полностью довериться поставщику услуги и дать ему возможность контролировать движения своих средств. Поскольку во многих странах процессуальное регулирование криптовалютной сферы ещё довольно несовершенное, пользователь может попасть в очень сложную ситуацию в случае мошенничества (или серьезной технической уязвимости) на стороне сервиса. Во-вторых, за ввод и вывод средств на кошельки этих сервисов с пользователей взимается комиссия. В-третьих, данные сервисы не позволяют осуществлять хостинг их программного обеспечения на серверах пользователей, вынуждая платить за размещение ПО на их собственных машинах.

Перечисленное выше ПО является проприетарным, поэтому какое-то его переиспользование не представляется возможным. Существуют, впрочем, несколько проектов с открытым исходным кодом. Главным их недостатком на данный момент является тот факт, что они поддержи-

вают не очень большое количество криптовалют. В то же время, конкретная реализация этой небольшой группы уже поддержанных блокчейнов накладывает интерфейсные и функциональные ограничения, с которыми придётся мириться, если захочется использовать какой-то из проектов, что на практике часто сводит на нет всю выгоду от первоначального ускорения разработки.

Среди проектов с открытым исходным кодом отдельно стоит отметить Blockchain-Payment-System [1], разрабатываемый совместно студентами матмеха и компанией DSX [4]. Они провели свой собственный обзор и так же пришли к выводу, что самостоятельно написать шлюз эффективнее и проще, чем расширить какой-то из имеющихся. Однако, разрабатываемый ими интерфейс с самого начала был ориентирован на возможность использования в как можно большем количестве проектов, в то время как описываемый здесь шлюз создавался для взаимодействия со специфическим сервисом, накладывающим жёсткие ограничения на технические решения. В силу данного обстоятельства эти две работы велись независимо друг от друга.

3.2. Обзор ключевых понятий

Блокчейн – это децентрализованная база данных особого вида, состоящая из записей, называемых блоками, которые связаны друг с другом средствами криптографии. Для того, чтобы новый блок попал в блокчейн, нужно, чтобы его одобрила большая часть участников сети. Процесс одобрения блока определяется алгоритмом консенсуса. Криптовалюты Bitcoin и Monero используют алгоритм, называемый Proof of work [13], суть которого состоит в том, что для создания нового блока необходимо решить вычислительно сложную задачу. Первый из участников сети, кому удалось найти правильное решение (оно проверяется остальными участниками), получает возможность создать блок и забирает награду за потраченные ресурсы. Но существуют и другие алгоритмы консенсуса. Все они устроены таким образом, чтобы создание нового блока было дорогой операцией. Это необходимо для обеспе-

чения устойчивости блокчейна к модификации данных: каждый блок содержит в себе свою хэш сумму и хэш сумму предыдущего блока, при изменении содержимого блока, его хэш сумма изменится, и для поддержания консистентности блокчейна необходимо будет поменять содержимое следующего блока, что в свою очередь изменит его хэш сумму. Таким образом, для изменения некоторого конкретного блока придётся поменять не только его, но и все блоки, идущие за ним. Поскольку создать даже один блок дорого, для значительного изменения блокчейна требуется настолько большое количество ресурсов, что на практике это почти невозможно осуществить.

Пользователь сети может сгенерировать пару из приватного и публичного ключей. Публичный ключ используется, чтобы получить из него адрес, на который можно переслать деньги. Приватный ключ используется, чтобы деньги, лежащие на некотором адресе, можно было потратить.

Основное предназначение блоков в сетях Monero и Bitcoin – хранить информацию о транзакциях между пользователями сети. Транзакция – это данные о переводе криптовалюты с одного адреса на другой. В этих валютах каждая транзакция содержит некоторые метаданные, необходимые для функционирования блокчейна, а также набор входов, выходов и количество переводимых денег (в других криптовалютах транзакции устроены иначе, но они не рассматриваются в данной работе, поэтому, чтобы не перегружать текст слишком большим количеством подробностей, здесь их описание не приводится). Входы – это структуры данных, ассоциированные с адресами, с которых переводятся деньги, с некоторыми балансами на них. Выходы – структуры данных, ассоциированные с адресами, на которые переводится желаемая сумма денег. Каждый выход некоторой транзакции в дальнейшем можно использовать как вход некоторой другой транзакции. Баланс некоторого адреса – это сумма балансов всех непотраченных выходов на этот адрес. Если сумма входов превышает количество, которое посылается на адрес назначения, остаток сформируется в отдельный выход и будет перечислен на указанный адрес сдачи. За создание транзакции сетью

может взиматься комиссия, которая выплачивается участнику, создавшему блок с этой транзакцией.

3.3. Unlinkable payments

В этом параграфе изложение ведётся с опорой на [15].

Во многих криптовалютах (например, Bitcoin) адрес пользователя становится уникальным идентификатором, по которому можно связать несколько транзакций друг с другом. В таких блокчейнах обеспечить полную независимость транзакций с одинаковым получателем можно лишь одним способом: использовать каждый адрес назначения только один раз. А это значит, для каждой транзакции получателю необходимо генерировать новый адрес назначения. Это непрактично, поэтому в Monero используется другой подход. В этой криптовалюте каждый адрес назначения особым образом генерируется из адреса получателя и некоторых случайных данных отправителя. Таким образом, если отправитель будет генерировать разные данные для разных транзакций, адреса назначений будут разными, и транзакции нельзя будет связать друг с другом.

Для понимания реализации Monero необходимо подробнее остановиться на деталях описанного выше алгоритма. Ниже приведен поэтапный процесс посылки транзакции в сети Monero.

Пусть Алиса хочет отправить платеж Бобу.

1. Боб публикует свой адрес (пару публичных ключей (A, B)).

2. Алиса генерирует случайное число r и вычисляет одноразовый публичный ключ $P = Hs(rA)G + B$.

3. Затем она использует P в качестве ключа для выхода транзакции и запаковывает значение $R = rG$ внутри неё рис. 1.

4. Алиса отправляет транзакцию.

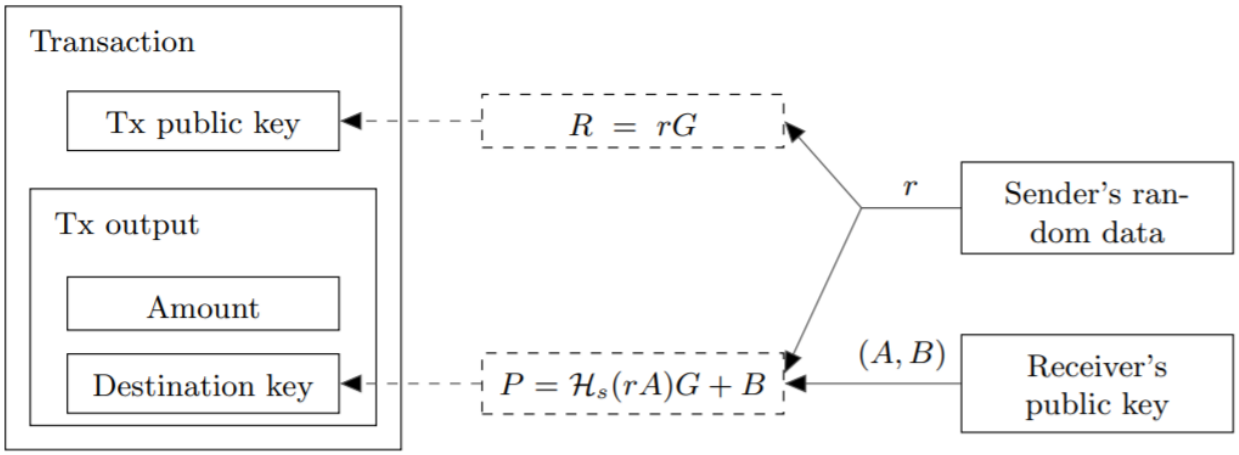

Рис. 1: Схематичная структура транзакции (источник: [15])

5. Боб проверяет каждую транзакцию нового блока парой своих приватных ключей (a, b), вычисляя $P' = Hs(aR)G + B$. Если Боб является получателем транзакции, то $aR = arG = rA \Rightarrow P = P'$

6. Боб понимает, что транзакция предназначается ему, и восстанавливает одноразовый приватный ключ $x = Hs(aR) + b$, такой, что $P = xG$.После этого он может потратить данный выход, подписав свою транзакцию ключом x рис. 2.

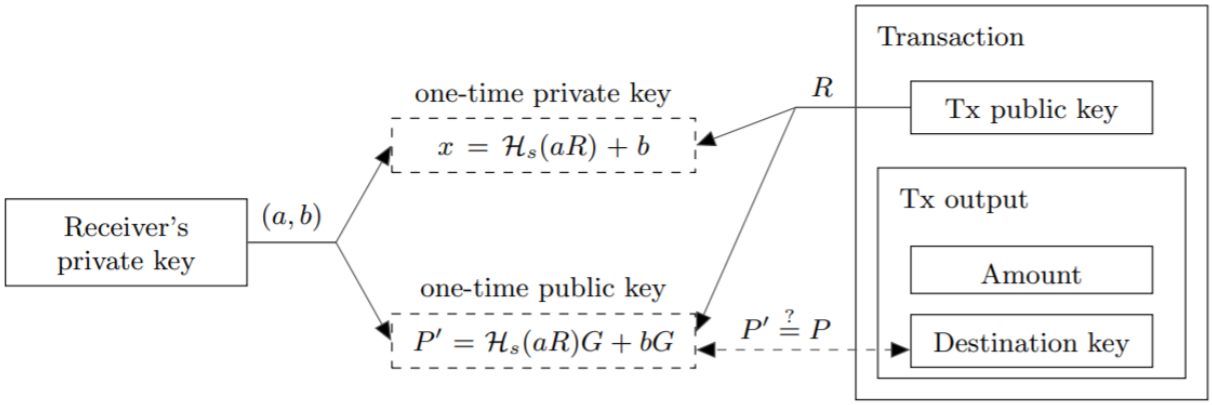

Рис. 2: Проверка входящей транзакции (источник: [15])

4. Реализация

В этом разделе содержится информация об архитектуре системы, представлен интерфейс шлюза и подробное описание его методов, а также приведены некоторые детали реализации конкретных криптовалют.

4.1. Архитектура

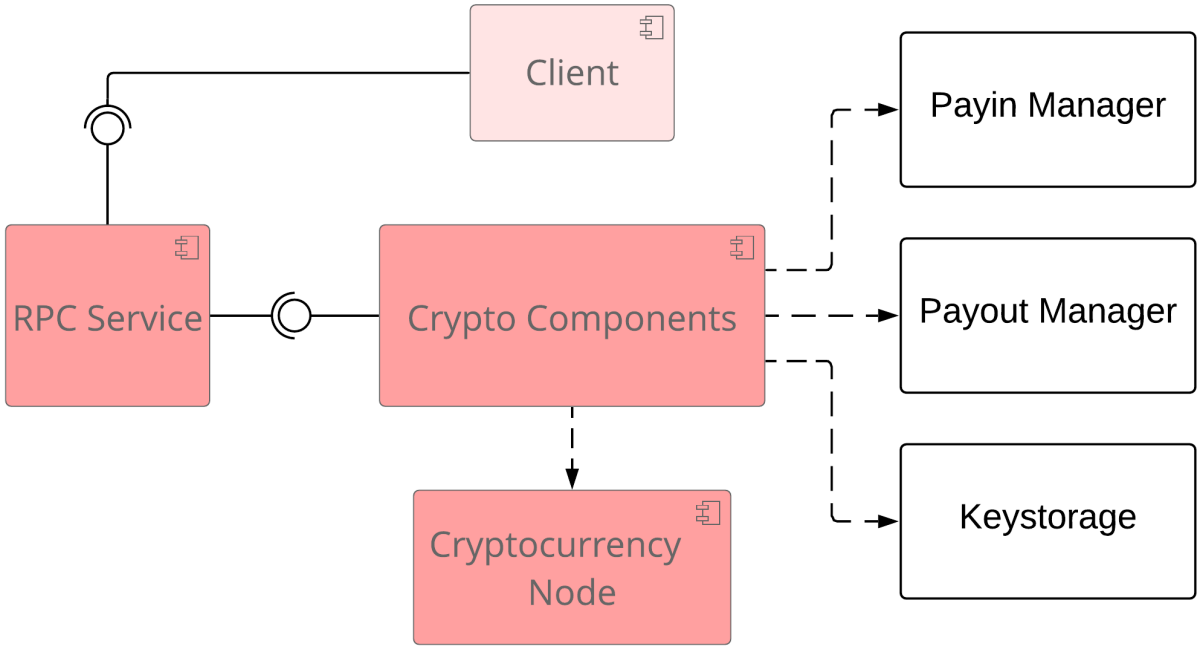

Рис. 3: Диаграмма компонентов шлюза

Выше представлена диаграмма основных компонентов проекта. Модуль CryptoComponents содержит в себе непосредственно логику шлюза, которая для каждой из криптовалют реализуется через три основных класса: PayinManager, PayoutManager и Keystorage. Для получения информации о транзакциях и отправления транзакций в блокчейн модуль CryptoComponents обращается к криптовалютному узлу по json-rpc. Модули RPCService и Client реализуют необходимые классы для инкапсулирования шлюза в самостоятельное приложение.

Главной задачей шлюза является упрощение работы внешнего сервиса с разными криптовалютами. Каждая из них имеет свой собствен-

ный формат адресов, схему учёта средств, алгоритм генерации комиссии и тому подобное. С точки зрения финансового сервиса всё это лишние детали. Ему важно знать только про наличие некоторых логических аккаунтов и движении денег между ними. Работа шлюза заключается в том, чтобы принять от сервиса запрос в обобщенном унифицированном виде, преобразовать его в данные конкретного блокчейна, сделать запрос в криптовалютный узел, получить ответ, унифицировать его и отдать обратно сервису. При такой схеме внешний сервис взаимодействует с каждой криптовалютой абсолютно одинаковым образом. Рассмотрим подробнее, как именно это реализовано.

4.1.1. Keystorage

Прежде всего, нам необходимо перейти от адресов блокчейна разного формата к абстрактным логическим аккаунтам сервиса, которые будут выглядеть одинаково для каждой криптовалюты. Для финансового сервиса с большим количеством пользователей естественным решением кажется сопоставить каждому адресу пару чисел — (account, wallet). Account — уникальный идентификатор каждого пользователя. Поскольку у каждого пользователя может быть много разных кошельков, идентификатор wallet необходим для указания конкретного номера кошелька. В этой реализации сервис на своей стороне будет работать с парами чисел, а шлюз проводить взаимно однозначное сопоставление между парой и реальным адресом.

Задача класса Keystorage состоит ровно в этом. Он определяет биективное отображение между парами чисел и адресами блокчейна. Поскольку адрес напрямую связан с приватным ключом, классу необходимо хранить/уметь генерировать ключи, этим и обусловлено его название. Интерфейсное описание Keystorage на этом заканчивается, он состоит из одной единственной функции.

```
Сигнатура:
std::vector<uint8_t>
getAddress(
    uint32_t account,
    uint32_t wallet
)
```

Примечание: сигнатуры, приведённые здесь, не являются точной копией сигнатур из реализации, они лишь призваны помочь лучше понять назначение методов. В частности, кое-где параметры передаются по значению только потому, что константные ссылки занимают больше места в тексте и затрудняют красивое единообразное отображение кода.

Далее будет описана реализация Keystorage, использованная для требуемых криптовалют. Нужное нам сопоставление можно осуществить множеством разных способов. Самый очевидный из них — просто сохранять отношение в какую-нибудь базу данных. Но существует более изящный подход. Криптография довольно большого класса валют позволяет построить подобное отображение вручную. Обычно это делается в виде специального дерева ключей, каждый новый уровень которого выводится из предыдущего. Получается конструкция, которая инициализируется некоторой последовательностью байт (она называется сидом, англ. seed), а потом, в зависимости от неё, детерминировано выдаёт по входным данным некоторый фиксированный ключ (из которого однозначно выводится конкретный адрес). Это называется детерминированным кошельком [14]. Используя его, достаточно сохранить лишь небольшой сид, после чего каждый раз при инициализации шлюза отдавать этот сид кошельку и получать структуру данных, которая однозначно выводит по кортежу чисел некоторый конкретный адрес сети. Все криптовалюты, с которыми взаимодействует сервис, поддерживают такую схему работы, поэтому было решено реализовать Keystorage именно через неё. Секретный сид хранится на стороне сервиса и каждый раз отдается в Keystorage при его инициализации. Если алгоритм генерации новых логических аккаунтов для пользователей реализует-

ся каким-нибудь псевдослучайным способом, с помощью одного только сида мы получаем возможность восстановить информацию обо всех транзакциях сервиса, в случае потери данных о них.

Нужно отдельно отметить, что конечно, вывод адреса из пар чисел занимает некоторое ненулевое время. Но это время пренебрежимо мало по сравнению со временем выполнения операций шлюза (подписи транзакций, например), поэтому никаких дополнительных ограничений эта реализация не накладывает. Если же когда-нибудь всё же потребуется свести это время к нулю, достаточно будет просто поддержить кэш адресов внутри Keystorage.

4.1.2. Managers

Прежде чем перейти к описанию менеджеров, стоит отметить несколько важных моментов. Все методы, инициирующие долгую операцию, разбиты на пары.

Первый метод пары передает аргументы, которые сохраняются в очередь. Каждой "долгой" операции менеджера соответствуют свои входная и выходная очереди. У многих методов есть один общий аргумент – userdata. Это специальный идентификатор каждого конкретного запроса. Когда результат операции складывается в выходную очередь, вместе с ним туда складывается userdata, чтобы внешний сервис мог определить по ней, результат какого именно запроса он получил. Внутри менеджера есть отдельный поток, который извлекает аргументы из входных очередей, выполняет нужную операцию и складывает её результаты в выходную очередь. Все "долгие" операции шлюза выполняются в этом отдельном потоке. Все очереди обрабатываются последовательно друг за другом. Внутри каждой из очередей операции тоже выполняются последовательно, одна за одной. К очередям осуществляется доступ из нескольких потоков, поэтому в коде используются примитивы синхронизации, чтобы избежать состояний гонки. Производительность такого способа обработки операций на данный момент является приемлемой. Если когда-нибудь потребуется её повысить, это легко осуществляется добавлением нескольких дополнительных потоков, каждый из которых

будет обрабатывать запросы независимо.

Второй метод пары извлекает из выходных очередей результаты операций и возвращает их наружу. Все эти методы начинаются с префикса “poll”. Если в выходной очереди слишком много сообщений, извлечены будут не все, а только часть из них, это необходимо, чтобы не возникало переполнений буфера. Поскольку размер каждого сообщения в байтах ограничен сверху, переполнения всегда можно избежать. Можно было бы ограничить количество забираемых сообщений еще и по времени работы метода, чтобы не выходить за рамки, определяемые сервисом, но в действительности, буфер переполняется быстрее, чем заканчивается время, поэтому этого делать не требуется. Модель polling’а была реализована потому, что внешний сервис хочет сам управлять своей нагрузкой и этот вариант предпочтительнее событийной модели.

Если в процессе работы менеджера происходит критическая ошибка (например, сервис пытается вывести деньги, которые должны лежать на его адресах, а кошелёк или криптовалютный узел возвращает ошибку “Not enough money”), он пишет об этом в логи и останавливается.

Кроме всего прочего, кое-какие детали реализации существенным образом отличаются в разных валютах. Для полноты изложения, чтобы обосновать возможность применения интерфейса в разных валютах или нагляднее показать мотивировку технических решений, в тексте специально приводятся некоторые соображения по реализации желаемой функциональности в блокчейнах, не рассматриваемых подробно в данной работе.

4.1.3. PayinManager

Задача этого класса – получение информации о новых поступлениях средств на адреса сервиса. Он определяет следующие методы.

AddToWatchlist

```
Сигнатура:
void
addToWatchlist(
    uint32_t account,
    uint32_t wallet,
    uint64_t userdata
)
```

Запрос на добавление логического адреса в список отслеживания. Сохраняет аргументы во входную очередь и завершает работу. Далее обработчик достаёт аргументы из очереди, выводит адрес из пары account/wallet с помощью Keystorage, добавляет его в свой список отслеживания и складывает копию в выходную очередь. Криптовалютный узел/кошелёк опрашивается на предмет наличия новых транзакций на любой адрес из списка, с периодичностью, которая передается в аргументах менеджера при инициализации. В аргументах также передается число блоков, после которого транзакция считается подтвержденной. До тех пор, пока заданное количество блоков не возникло в блокчейне, менеджер не будет сообщать о транзакции сервису.

Также менеджер имеет специальный метод, позволяющий выставлять блок, начиная с которого он сканирует блокчейн. Это необходимо для того, чтобы при перезапуске сервиса повторно выдавать информацию обо всех интересующих сервис транзакциях. О транзакциях, пришедших на адрес до указанного блока, сообщаться не будет.

На данный момент этой реализации достаточно для успешного функционирования сервиса. Но если вдруг из-за асинхронности интерфейса возникнет проблема с тем, что пользователям слишком долго придется ждать свой реальный адрес на фронтенде приложения, довольно легко поддержать некоторый пул уже добавленных в список отслеживания адресов, из которого можно мгновенно выдавать новому пользователю готовый адрес.

Процесс получения данных о новой транзакции изображен на одной

из диаграмм рис. 6 в приложении.

PollAddToWatchlistReponses

```
Сигнатура:
std::vector<AddToWhatchListResponse>
pollAddToWatchlistReponses()
```

Сбор ответов на все выполненные запросы addToWatchlist.

GetBalance

```
Сигнатура:
void
getBalance(
    uint32_t account,
    uint32_t wallet,
    uint64_t userdata
)
```

Запрос баланса аккаунта. Складывает аргументы во входную очередь и завершает работу. Обработчик достаёт аргументы из очереди, с помощью Keystorage генерирует реальный адрес блокчейна и запрашивает баланс у кошелька/криптовалютного узла. Полученное значение складывается в выходную очередь.

PollGetBalanceResponeses

```
Сигнатура:
std::vector<CompletedBalance> pollGetBalanceResponeses()
```

Сбор ответов на все выполненные запросы getBalance.

PollIncomingTransfers

```
Сигнатура:
std::vector<IncomingTransfer> pollIncomingTransfers()
```

Сбор информации о зачислении новых средств. IncomingTransfer состоит из пары account/wallet, на которую совершен платеж, количества поступивших средств и идентификатора транзакции. Если в одной и той же транзакции было получено несколько разных выходов на один и тот же адрес, менеджер сформирует несколько сообщений с одинаковым идентификатором.

SetReportingStartHeight

Сигнатура:

```
void setReportingStartHeight(uint64_t height)
```

Блокирующий метод, устанавливающий высоту блокчейна, начиная с которой менеджер ищет транзакции сервиса.

4.1.4. PayoutManager

Последним фрагментом шлюза является функциональность по выводу денег сервиса. За всё, что связано с посылкой и генерацией транзакций, отвечает класс PayoutManager. Следует прокомментировать, что тип uint256_t не является стандартным типом C++. Он применяется, потому что его использование необходимо для некоторых других валют, не входящих в рассмотрение в данной работе. При его реализации была использована одна из библиотек для работы с большими числами.

Итак, методы класса выглядят следующим образом.

EstimateFee

```
Сигнатура:
void
estimateFee(
    uint256_t amount,
    std::vector<std::pair<uint32_t, uint32_t>> inputAddresses,
    std::string destination,
    uint64_t userdata
)
```

Запрос на оценку комиссии. Этот метод выдает верхнюю оценку комиссии для транзакции, которую пришлось бы заплатить, чтобы вывести запрашиваемую сумму в данный момент (при текущем состоянии блокчейна). Реализация, обычно, заключается в том, чтобы сформировать транзакцию, и, не отправляя её в сеть, посмотреть, сколько за нее придется заплатить. Для случаев, когда перевод связан с вызовами смарт-контракта (например, ERC20 [7] токены на основе Ethereum [6]), оценка будет заключаться в том, чтобы выполнить транзакцию локально на имеющемся в распоряжении криптовалютном узле. Понятно, что оценку не всегда возможно сделать точной (она напрямую зависит от конкретной криптовалюты и текущих балансов адресов), и при изменении состояния блокчейна сумма может отличаться. Если количество неудачных оценок на практике окажется слишком велико, предлагается эмпирическим образом подобрать некоторый повышающий коэффициент, чтобы довести их до удовлетворительного уровня.

Используется метод для того, чтобы сервис имел хотя бы примерное представление о сумме, которую ему нужно будет потратить за перевод. При желании посмотреть наглядную демонстрацию процесса вывода денег, который начинается с вызова данного метода, можно обратиться к соответствующей диаграмме (рис. 5) из приложения. Оценка будет передана в качестве аргумента при дальнейшей реальной отправке и, если после генерации уже финальной транзакции выяснится, что ее комиссия больше оценки, выводу присвоится статус неудачного и в выход-

ную очередь сложится соответствующая ошибка. Опять же, в случае смарт-контрактов, реальная комиссия становится известна только после попадания блока в блокчейн. К сожалению, это ограничение никак не обойти. Но можно оценить комиссию описанным выше способом ещё раз, перед отправкой, чтобы повысить шансы, что комиссия уложится в необходимый диапазон. В случае, если комиссия всё же окажется недостаточной, в выходную очередь сложится результат со специальной ошибкой. Также следует отметить, что в таких ситуациях придется дополнительно какое-то время опрашивать кошелек/криптовалютный узел на предмет того, попала ли транзакция в блокчейн, а потом ещё ждать какое-то количество подтверждений (скорее всего, передаваемое параметром при инициализации менеджера).

Комиссия возвращается в единицах той же криптовалюты, что и amount. Известно, что существуют криптовалюты, в которых единицы переводимой криптовалюты и комиссии отличаются. Это справедливо, например, для тех же ERC20 токенов. Для необходимого нам варианта использования данного метода, в этом случае, чтобы сохранить семантическую целостность интерфейса, достаточно будет представить полученную комиссию в виде реального значения, умноженного на некоторую константу, обозначающую условную цену одной из криптовалют в единицах другой.

Метод принимает сумму вывода, множество логических адресов, балансы которых участвуют в наборе денег для перевода, и адрес назначения. Он сохраняет все эти аргументы во входную очередь и завершает работу. Далее аргументы извлекаются в обработчике, происходит генерация транзакции, и оценка комиссии, требуемой для отправки этой транзакции, помещается в выходную очередь. В случае ошибки на любом этапе работы, запрос считается неудачным и в выходную очередь, вместо значения комиссии, сохраняется сообщение об ошибке.

PollEstimateFeeResponeses

Сигнатура:

```
std::vector<EstimatedFee> pollEstimateFeeResponeses()
```

Сбор ответов на все выполненные запросы estimateFee.

SendTx

```
Сигнатура:
std::vector<uint8_t> sendTx(
    uint256_t estimatedFee,
    uint256_t amount,
    std::vector<std::pair<uint32_t, uint32_t>> inputAddresses,
    std::string destination,
    uint64_t userdata
)
```

Запрос на вывод денег с адресов сервиса, принимает те же параметры, что и estimateFee и, дополнительно, оцененную ранее с помощью этого метода комиссию. Менеджер сохраняет аргументы в контейнер, а затем генерирует некоторый контекст, который ассоциирован с транзакцией, и возвращает его наружу. Поскольку все контексты генерируются в одном потоке, никакие примитивы синхронизации не используются. Сами контексты выбираются таким образом, чтобы их можно было сгенерировать быстро (достаточно быстро, чтобы внешний сервис эта скорость устраивала). Эти два факта позволяют сделать данный метод блокирующим.

Назначение контекста следующее: если после отправки транзакции в криптовалютный узел/кошелёк ответ не был получен, и произошла какая-нибудь критическая ошибка, в результате которой шлюз аварийно завершил работу, у внешнего сервиса не будет никакой информации о том, выполнилась транзакция или нет. Поскольку сервис обязан знать обо всех потраченных деньгах, эту информацию нужно как-то восстановить. Для этого каждый вывод денег связывается с уникальным контекстом, по которому специальный метод PayoutManager'а сможет восстановить статус транзакции. Алгоритм генерации контекста и сам вид контекста могут быть разными у разных криптовалют. В рассматриваемых нами реализациях контекст никак не связан с конкрет-

ными аргументами, передаваемыми на вход методу. Но поскольку для какой-нибудь из будущих криптовалют это может потребоваться, вместо простого getNextContext(), который ничего не принимает, решено было спроектировать интерфейс именно так.

Может возникнуть вопрос, почему интерфейс изначально не был спроектирован таким образом, чтобы в качестве контекста выступал просто идентификатор транзакции. Технически, это действительно сделать можно, достаточно просто добавить ещё один poll-метод (так как генерация транзакция – долгая операция). Однако, при полностью асинхронной генерации и отправке транзакций, необходимо отслеживать, чтобы на уровне блокчейна не возникало попыток потратить одни и те же деньги несколько раз, с нескольких сгенерированных (но не отправленных) подряд транзакций. Не все криптовалютные узлы по умолчанию реализуют всю необходимую функциональность, позволяющую это сделать. Например, в кошельке Monero, (на момент написания кода шлюза, по крайней мере), её не было. То есть, пришлось бы поддерживать это самостоятельно для каждой такой валюты. Этого делать не хотелось, поэтому решили остановиться на собственных контекстах, которые, с точки зрения программистов, реализовать было проще.

ConfirmSend

`Сигнатура:`

```
void confirmSend(uint64_t userdata)
```

Вызов этого метода подтверждает, что сервис успешно получил от шлюза контекст транзакции. По userdata шлюз находит необходимые аргументы и перемещает их во входную очередь для отправки, после чего метод завершает работу. Далее аргументы извлекаются в обработчике, генерируется транзакция, производится сравнение с оценкой комиссии и, если всё хорошо, транзакция отправляется в сеть. Менеджер контролирует набор входных адресов и выбор адреса сдачи (если она присутствует). После отправки менеджер дожидается момента, когда становятся известны и идентификатор транзакции, и реальная

комиссия, и складывает эти значения в выходную очередь.

Важным моментом является тот факт, что иногда (несколько раз в году) происходит резкий скачок курса криптовалюты. В работе с биткойном это может привести к тому, что размер средней комиссии резко возрастает. Поскольку майнеры отдают предпочтение транзакциям с более высокой комиссией, транзакции отправленные до скачка курса в таком случае, будут долгое время висеть не выбранными. На данный момент интерфейс не предусматривает никакой специальной обработки этих ситуаций, и в реализации просто используются значения комиссий выше средних. Этот случай не обрабатывается, поскольку он довольно редкий, и на данный момент не хочется тратить на него время. Но понятно, что как только его нужно будет поддержать, достаточно будет использовать RBF транзакции, описанные в Bip-0125 [5].

PollConfirmSendResponses

Сигнатура:

```
std::vector<CompletedSend> pollConfirmSendResponses()
```

Сбор ответов на все выполненные запросы sendTx.

RefreshTxs

Сигнатура:

```
void refreshTxs(std::vector<Context> contexts)
```

Метод, используемый после аварийного завершения работы. Он принимает множество контекстов и складывает во входную очередь. Обработчик достаёт аргументы из очереди и сканирует блокчейн на предмет наличия транзакций, связанных с этими контекстами. Результаты складываются в ту же очередь, что и результаты запросов confirmSend, и становятся доступны сервису через метод pollConfirmSendResponses. Структура Context содержит в себе контекст из sendTx и userdata. Наглядно работу метода можно посмотреть на диаграмме в приложении (рис. 4)

ValidateAddress

```
Сигнатура:
bool validateAddress(std::string address)
```

Метод является блокирующим, поскольку выполняется он очень быстро. Принимает строку и возвращает истину или ложь, в зависимости от того, является ли строка корректным адресом реализуемого блокчейна.

4.1.5. RpcService и Client

Модули, содержащие классы, осуществляющие внешнюю обертку шлюза. С целью уменьшения количества ошибок в коде коммуникации и упрощения добавления новых методов, был написан плагин к Protobuf [11], который позволяет автоматически сгенерировать большую часть этого кода. По декларативному описанию он генерирует сообщения протокола, методы их сериализации/десериализации, а также некоторый, довольно однотипный, код обертки.

По существу, руками остается написать только пару функций, которые отвечают непосредственно за механизм транспорта, одну на серверной стороне, вторую на клиентской. В них поддерживается конкретный вид коммуникации (очереди, сокеты, пайпы), а сгенерированные методы Protobuf'а используются, чтобы получить из сообщений сырые байты и передать их по транспортной системе. Этот транспорт-агностик подход дает возможность очень быстро переключаться между разными способами передачи сообщений, выбирая оптимальную для конкретной конфигурации сервиса и шлюза, а также очень быстро поддержать любой новый способ, который может потребоваться в будущем.

4.1.6. Тестовое окружение

Это не является напрямую частью архитектуры, но стоит отметить, что во время разработки было реализовано отдельное тестовое окружение с использованием GTest [8]. Были написаны компоненты, эмулирующие поведение криптовалютных узлов, и с их помощью протестирова-

ны различные нехорошие сценарии, чтобы убидиться, что шлюз умеет корректно их обрабатывать.

4.2. Детали реализации Monero

В качестве криптовалютного узла, с которым работает шлюз, был выбран официальный узел Monero, написанный на C++ [9]. Конкретнее, шлюз взаимодействует с одним из его компонентов – rpc кошельком, который пришлось частично переписать, чтобы удовлетворить нашим требованиям.

Во-первых, оказалось, что в официальном узле Monero существует поддержка детерминированного кошелька, который генерирует множество адресов из заданного сида. Но этот кошелек не умел принимать сид снаружи, он лишь создавал его самостоятельно и выдавал пользователю в специальном формате. Этот код был переписан, чтобы можно было открывать детерминированный кошелёк, передавая ему бинарный сид по сети. Когда обнаружилось, что детерминированность ключей в Monero уже есть, в качестве MVP было решено оставить Keystorage пустым и всю работу с ключами делегировать именно кошельку, поэтому в него просто были добавлены rpc методы создания новых адресов. Оставить Keystorage пустым было решено еще и потому, что если уж переносить криптографию из кошелька в шлюз, то делать это стоило бы с *очень* глубоким рефакторингом кода. К сожалению, по мнению разработчиков, (разумеется, это субъективная оценка) существенная часть кода в репозитории Monero довольно плохого качества, и если брать на себя ответственность за его поддержку, то сначала нужно его переписать, чтобы он стал понятным. На данный момент на это просто нет времени. Но в дальнейшем, для безопасности, всё же планируется перенести его в Keystorage.

Во-вторых, был модифицирован алгоритм генерации транзакции. Как уже было указано ранее, потребовалось внедрить в транзакцию какой-нибудь маркер, чтобы можно было с его помощью искать транзакции в блоках, и при этом сделать это так, чтобы транзакция не

отличалась от других транзакций в сети (то есть, просто записывать побочные данные было нельзя, тогда транзакции шлюза оказались бы похожи друг на друга, что уменьшает степень анонимности). Идеальное решение оказалось основанным на принципах алгоритма, описанном в параграфе Unlinkable payments. Чтобы добиться нужной функциональности, достаточно использовать в качестве r не случайные данные, а некоторый сид, сгенерированный шлюзом, который он отдаёт сервису ещё до того, как послал запрос в кошелек. Этот сид и является контекстом в Monero. При перезапуске, контексты неизвестных транзакций складываются в refreshTxs, он отправляет их в криптовалютный узел, а тот вычисляет публичные ключи транзакций и проверяет, существуют ли в блокчейне транзакции с данными ключами.

Как описано выше, Keystorage на данный момент в Monero не используется. Текущая структура Payin и Payout менеджеров тоже получилась достаточно простой.

PayinManager при старте запускает отдельный поток, в котором кошелёк, с заданной периодичностью, опрашивается о новых входящих транзакциях на список отслеживаемых адресов. Все новые транзакции складываются в очередь, откуда потом забираются внешним сервисом. При добавлении новой пары в список отслеживания, он просто посылает кошельку запрос и возвращает адрес, который кошелёк создал.

PayoutManager также практически не содержит сложной логики внутри себя и просто перенаправляет запросы кошельку. Главная функциональность, которую он реализует — валидация адресов, генерация контекстов для исходящих транзакций и восстановление корректного состояния системы после серьезной ошибки.

4.3. Детали реализации Bitcoin'а

Keystorage реализует алгоритм BIP32 [14]. Он принимает в конструктор сид и описанным в обзоре образом генерирует пары приватных и публичных ключей. Алгоритм был реализован самостоятельно, потому что на момент написания кода в свободном доступе не удалось

найти его простой реализации. Были найдены различные библиотечные варианты, использующие внутри себя множество ненужных оберток, логгеров, хелперов и вдобавок линкующихся с внутренними библиотеками проекта. Был сделан вывод, что написать свою версию (используя библиотечный код хеш-функций, в котором всех этих проблем не было) будет просто быстрее, чем аккуратно вытаскивать оттуда нужные части.

Компонент поддерживает сетевое соединение с проприетарным кошельком Bitcoin'a, в который были добавлены необходимые rpc методы. PayinManager импортирует в него ключи, сгенерированные в Keystorage. Генерация транзакции сервиса происходит в несколько этапов.

PayoutManager поочередно вызывает ряд rpc запросов у кошелька. Сначала запрашивается набор входов, чтобы покрыть необходимую сумму транзакции. Затем генерируется адрес для сдачи и делается запрос на создание сырой неподписанной транзакции. Потом сырая транзакция подписывается ключами из Keystorage и отдаётся кошельку, который отправляет её в сеть.

В сети Bitcoin, в отличие от Monero, в качестве входа одной транзакции можно использовать выход другой транзакции, которая содержится с первой в одном и том же блоке. Это даёт возможность выстроить из транзакций связный список, с гарантией, что если транзакция с номером N не попала в блокчейн, то все транзакции с номерами большими, чем N, так же в него не попадут. Собственно, на этом и основан алгоритм восстановления корректного состояния сервиса после аварийной остановки. Каждая новая транзакция использует выход сдачи предыдущей транзакции в качестве одного из своих входов. Адреса сдачи имеют логические адреса (0, n). Контекстом транзакции выступает второй элемент пары. Этот индекс уникален для каждой транзакции, индексы двух последовательных транзакций отличаются на единицу. Чтобы восстановить состояние сервиса, мы отдаём ему пачку контекстов транзакций, статусы которых мы не знаем, получаем из них адреса сдачи и проверяем, содержатся ли в блокчейне транзакции с выходом на адрес. Если транзакция есть, мы сообщаем о ней сервису. Если нет – останав-

ливаем алгоритм и помечаем текущую и все последующие транзакции как неотправленные.

Заключение

В рамках дипломной работы были получены следующие результаты.

1. Изучены принципы и детали функционирования ряда криптовалют, проведен анализ существующих решений.

2. Разработан интерфейс для собственного шлюза, написана его кроссплатформенная реализация для криптовалют Monero и Bitcoin.

3. Продуманы и реализованы механизмы восстановления системы после аварийного завершения работы.

4. Реализован rpc сервис, оборачивающий основное приложение.

5. Реализовано тестовое окружение, на котором был протестирован шлюз.

В дальнейшем планируется написать реализации интерфейса для других криптовалют, а также вынести всю работу с ключами из криптовалютных узлов непосредственно в шлюз, чтобы увеличить уровень безопасности.

Выражаю искреннюю благодарность Кириленко Якову Александровичу за своевременные консультации и помощь в ходе работы, а также Долголеву Филиппу Петровичу и компании “DSX Technologies Limited” за проведение рецензирования и советы по написанию текста.

Приложение

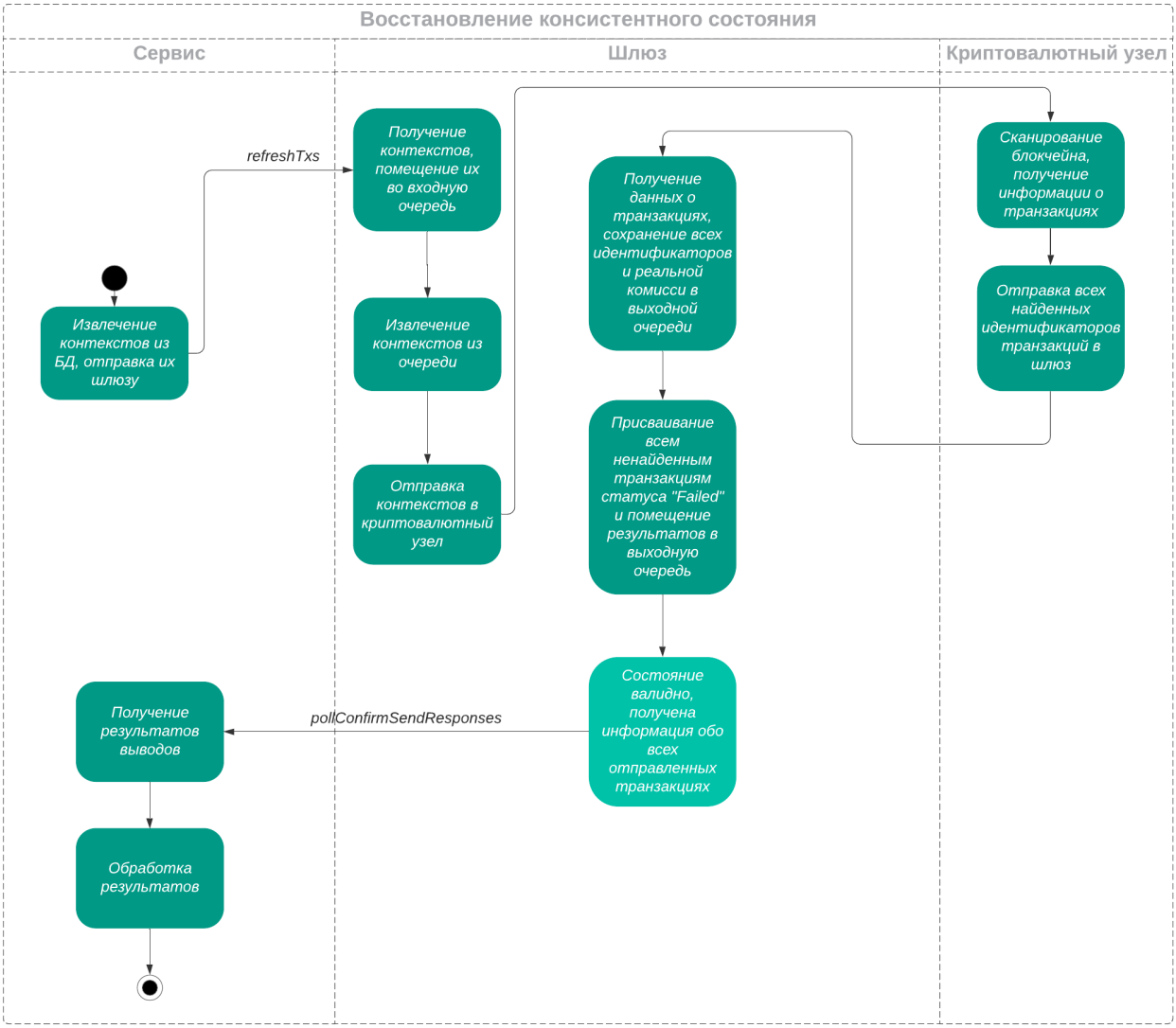

Рис. 4: Определение актуального статуса транзакций после аварийного завершения работы

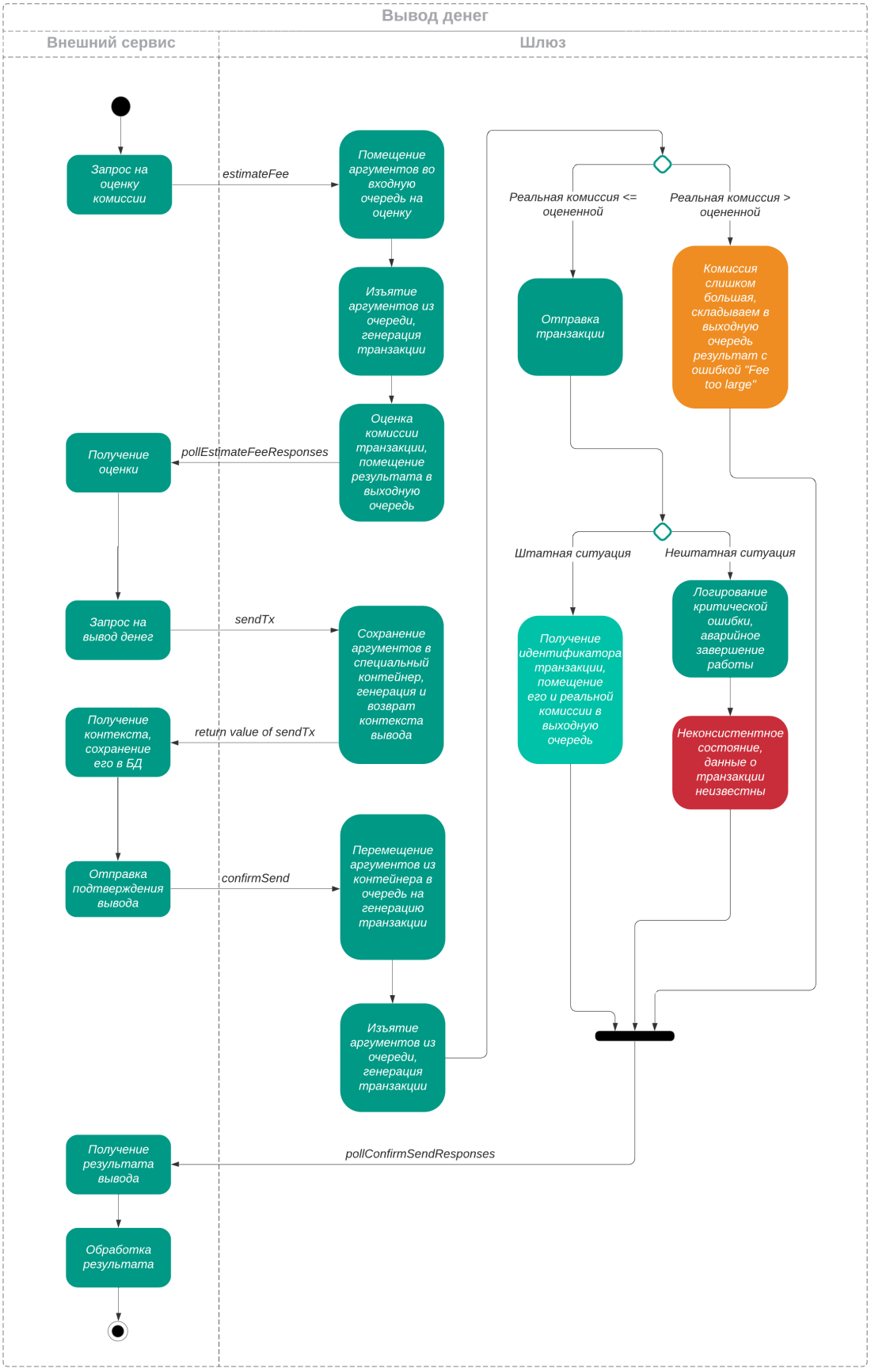

Рис. 5: Процесс вывода денег из шлюза

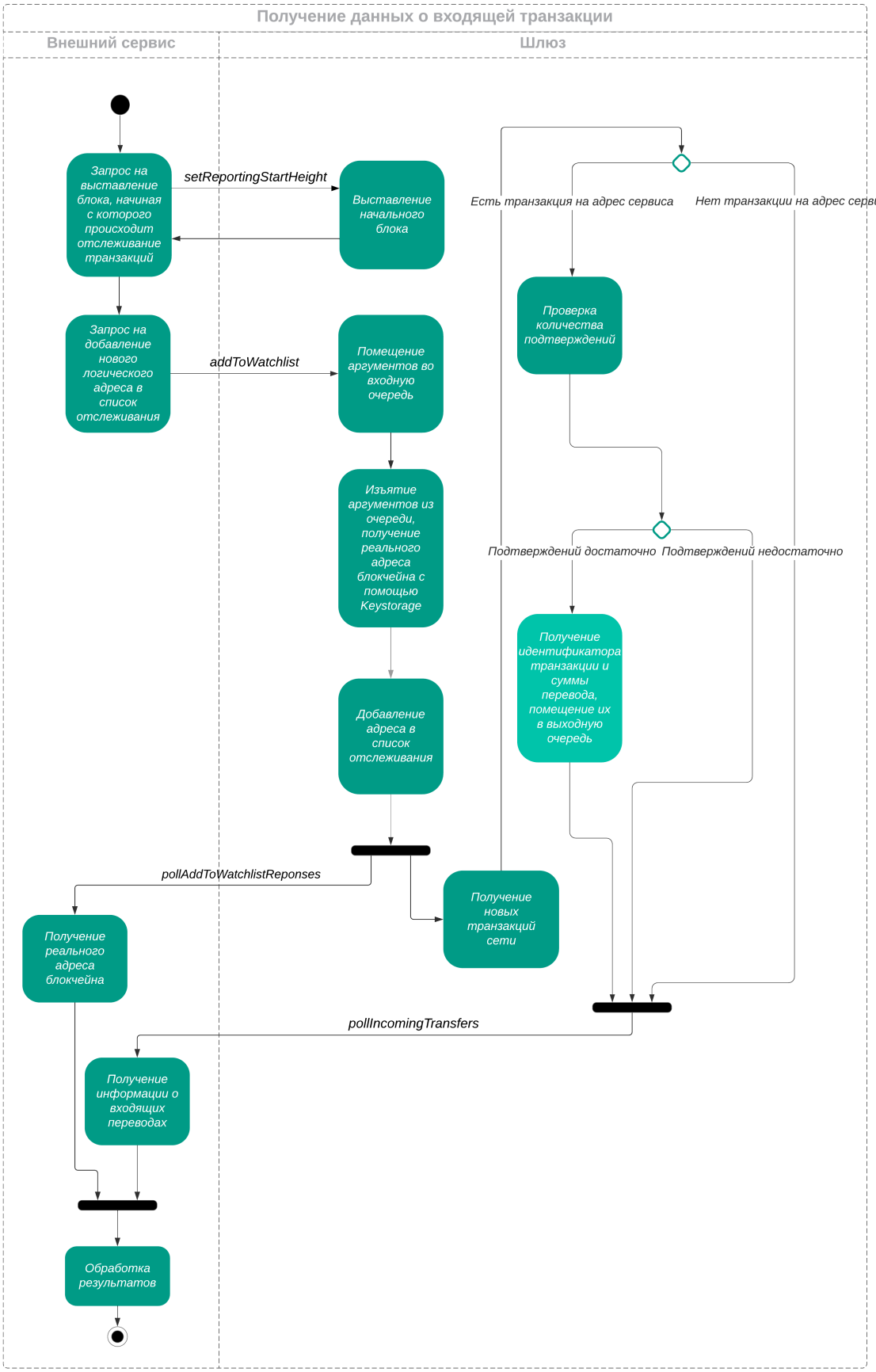

Рис. 6: Получение данных о входящей транзакции

Список литературы

[1] Blockchain payment system. — URL: https://github.com/dsx-tech/Blockchain-Payment-System (online; accessed: 04.06.2020).

[2] Bondcap. Internet trends. — 2019. — URL: https://www.bondcap.com/report/itr19/ (online; accessed: 04.06.2020).

[3] Coinmarketcap. — URL: https://coinmarketcap.com/all/views/all/ (online; accessed: 04.06.2020).

[4] DSX technologies limited. — URL: https://dsxglobal.com (online; accessed: 04.06.2020).

[5] David A. Harding Peter Todd. Opt-in Full Replace-by-Fee Signaling. — 2015. — URL: https://github.com/bitcoin/bips/blob/master/bip-0125.mediawiki (online; accessed: 04.06.2020).

[6] Ethereum. — URL: https://ethereum.org/ru/ (online; accessed: 04.06.2020).

[7] Fabian Vogelsteller Vitalik Buterin. ERC-20 Token Standard. — 2015. — URL: https://eips.ethereum.org/EIPS/eip-20 (online; accessed: 04.06.2020).

[8] GTest. — URL: https://github.com/google/googletest (online; accessed: 04.06.2020).

[9] Monero. — URL: https://github.com/monero-project/monero (online; accessed: 04.06.2020).

[10] Nakamoto Satoshi. Bitcoin: A Peer-to-Peer Electronic Cash System. — 2008. — URL: https://bitcoin.org/bitcoin.pdf (online; accessed: 04.06.2020).

[11] Protocol buffers. — URL: https://developers.google.com/protocol-buffers (online; accessed: 04.06.2020).

[12] Time. Here's Why PayPal Just Added a Record Number of New Users. — 2016. — URL: https://time.com/4197704/tech-paypal-users-earnings/ (online; accessed: 04.06.2020).

[13] Wikipedia. Proof of work. — URL: https://en.wikipedia.org/wiki/Proof_of_work (online; accessed: 04.06.2020).

[14] Wuille Pieter. Hierarchical Deterministic Wallets. — 2012. — URL: https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki (online; accessed: 04.06.2020).

[15] van Saberhagen Nicolas. CryptoNote v 2.0. — URL: https://cryptonote.org/whitepaper.pdf (online; accessed: 04.06.2020).