Изучите блокчейн, создав свой

Краткий путь к пониманию криптовалют

Дэниэл ван Флаймен

Apress®

Изучите блокчейн, создав свой

Краткий путь к пониманию криптовалют

Дэниэл ван Флаймен

Apress®

Изучите блокчейн, создав свой: краткий путь к пониманию криптовалют

Дэниэл ван Флаймен
Нью-Йорк, США

ISBN-13 (бумажное): 978-1-4842-5170-6 ISBN-13 (электронное): 978-1-4842-5171-3
https://doi.org/10.1007/978-1-4842-5171-3

Руководитель, Apress Media LLC: Welmoed Spahr
Редактор по приобретению: Шива Рамачандран
Производственный редактор: Рита Фернандо
Редактор-координатор: Рита Фернандо

Дизайн обложки eStudioCalamar

Распространяется в книжной торговле по всему миру компанией Springer Science+Business Media New York, 1 New York Plaza, New York, NY 10004. Телефон 1-800-SPRINGER, факс (201) 348-4505, e-mail orders-ny@springer-sbm.com, или посетите www.springeronline.com. Apress Media, LLC является California LLC а единственным участником (владельцем) является Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc – это корпорация **Delaware**.

За информацией о переводах обращайтесь по e-mail booktranslations@springernature.com; для перепечатки, мягкой обложки или прав на аудио по e-mail bookpermissions@springernature.com.

Книги Apress можно приобрести оптом для академического, корпоративного или рекламного использования. Для получения дополнительной информации посетите нашу веб-страницу массовых продаж печатных и электронных книг по адресу http://www.apress.com/bulk-sales.

Любой исходный код или другие дополнительные материалы, на которые ссылается автор в этой книге, доступны читателям на GitHub через страницу продукта по адресу www.apress.com/ 978-1-4842-5170-6. Для получения более подробной информации посетите http://www.apress.com/source-code.

Отпечатано на бумаге из переработанных материалов

Посвящается Джошуа,
который заканчивает начатое.

Содержание

Глава 7: Создание транзакционного узла

Об авторе

Дэниэл ван Флаймен в настоящее время является техническим директором компании Candid в Нью-Йорке. Будучи опытным программистом на Python, он регулярно пишет код для популярных проектов с открытым исходным кодом и является гостем подкаста Software Engineering Daily, участвуя в таких популярных эпизодах, как Understanding Bitcoin Transactions и Blockchain Engineering. Он часто пишет на Medium.com и опубликовал ряд популярных статей, таких как «Изучайте блокчейны, создавая их» и «Изучайте блокчейны, используя электронные таблицы».

О техническом рецензенте

Федерико Ульфо — многогранный инженер-программист и предприниматель, имеющий опыт создания крупномасштабных API и ETL. Он основал Lightning Network NYC и встречи Learning Bitcoin. Его интересы простираются от криптовалют до экономики, философии, садоводства и многих других тем. Вы можете связаться с ним на ulfo.it.

Слова благодарности

Эта книга посвящается моему брату Джошуа, который всегда доводит начатое до конца. Большое спасибо

- Моему другу и коллеге-преподавателю биткойнов, **Джастину Муну**, за помощь в прояснении концепций, тестировании кода и предоставлении разумных советов, когда это необходимо.

  Моему другу **Федерико Ульфо**, не только за тяжелую еженедельную работу по анализу и перепроверке моей рукописи, но и за участие со мной в бесчисленных мероприятиях и конференциях, посвященных Биткойну и Lightning за последние несколько лет.

- **Рите Фернандо** и **Шиванги Рамачандран** из Apress за то, что поверили в меня и сделали эту книгу реальностью.

Введение

Еще одна книга о блокчейнах? Почему?

Разобраться в блокчейнах непросто. Или, по крайней мере, не для меня: когда Биткойн впервые попал в новости, я попытался узнать, как он работает, и обнаружил, что слишком мало ресурсов адресовано программистам (таким как я). Всегда существовала справочная wiki о биткойнах (https:// en. bitcoin. it), но в те дни она не была так четко организована, как сегодня, и хотя я читал технический документ Сатоши, я сначала не очень понял его — по крайней мере, не так, как работали криптографические части.

Я блуждал по YouTube, изучая подробные руководства и чувствовал разочарование от примеров, которые не передавали концепции достаточно четко. Итак, я решил попробовать создать блокчейн самостоятельно и задокументировать все, что я узнал на этом пути. При этом я понял, почему криптовалюты так сложно объяснить и понять; это потому, что вам сначала нужно определить ингредиенты цифровых денег:

- Как создаются деньги? (Майнинг)
- Как Алиса отправляет деньги Бобу? (Транзакции с цифровой подписью)
- Кто отслеживает все эти транзакции и полученные деньги? (Все, через распределенный реестр)

Эти точки высокого уровня основаны на отдельных квантах знаний, которые необходимо понять, прежде чем их можно будет объединить в набор общепринятых правил, которым все следуют. И лучший способ понять эти разрозненные концепции — по частям — использовать их на практике для создания собственной

криптовалюты. Итак, я написал эту книгу для людей, которые испытывают то же разочарование, что и я, и преодолевают его, работая с предметом на уровне кода. Если вы будете следовать и делать то же самое, я уверен, что в конце этой книги вы будете иметь четкое представление о том, как это все работает.

Настроить себя на успех

Репозиторий GitHub

Окончательный код находится по адресу
https://github.com/dvf/blockchain-book.

Но попробуйте написать код самостоятельно — код структурирован таким образом, что методы заглушаются на высоком уровне, а детали заполняются постепенно. Этот код постоянно обновляется, поэтому он удобен как путеводная звезда.

Потратьте время на настройку среды разработки

Используйте хорошую IDE (интегрированную среду разработки), например Microsoft VSCode или JetBrains PyCharm. Они бесплатны и прекрасно обнаружат ошибки в вашем коде раньше вас. И стоит потратить время на настройку IDE перед тем, как начать. Тратьте свое время на беспокойство о блокчейнах, а не о синтаксических ошибках в вашем коде.

Знайте, где найти ответы

Просмотрите и задайте вопросы на странице *Issues* репозитория GitHub. У репозитория большое сообщество, поэтому вы, вероятно, встретите других с похожими проблемами. И если вы столкнетесь с ошибками или ошибками, я умоляю вас открыть *Issues*.

Не говорите на Python?

Это нормально. Python известен своей удобочитаемостью; это очень простой язык для понимания. Я видел, как другие программисты (C#, JavaScript и Rust) делали примеры из книги на лету.

ГЛАВА 1

Подготовка к разработке приложений

Для тех, кто не в курсе, Python — один из самых популярных языков. Он широко используется везде — от академических и научных кругов до крупных веб-приложений, таких как Instagram. Частично его популярность связана с множеством библиотек, пакетов и расширений, доступных бесплатно в Интернете, а также с простотой чтения из-за его сходства с псевдокодом.

В этой главе мы убедимся, что ваш компьютер правильно настроен для разработки приложений и правильно установлен Python. Затем я покажу вам, как создать практичный проект Python и как установить зависимости.

Версии Python

Python поставляется в двух вариантах: Версия 2 и Версия 3. Версия 2 больше не поддерживается Python Software Foundation, но она по-прежнему поставляется предустановленной в большинстве операционных систем, поскольку используется множеством внутренних инструментов. Другая сложность заключается в том, что разные операционные системы устанавливают Python в разные места файловой системы. Эти факторы усложняют настройку среды разработки.

Мы попробуем обойти эти препятствия путем установки и использования инструментов, помогающих нам управлять установками Python.

Примечание В качестве условного обозначения в этой книге мы будем использовать префикс команды терминала с помощью символа $. Вывод будет показан в виде обычного текста.

Установка Python

Установка в срде Windows

Python.org содержит загружаемые двоичные файлы для Windows. Посетите www.python.org/downloads/windows/ и скачайте двоичный файл дляPython 3.8.

После загрузки установите Python 3.8, обязательно выбрав опцию

- Удаления предыдущей версии Python.
- Установки pip (менеджера пакетов Python).
- Добавления Python в PATH (что позволит вам запускать Python из командной строки).

После установки, чтобы убедиться, что вы все сделали правильно, откройте командную строку и проверьте версию Python:

```
C:\Users\dan> python --version
3.8.3
```

Установка в среде macOS

Хотя macOS поставляется с версией Python для собственных целей, мы **не хотим изменять ее при разработке,** поэтому мы будем устанавливать свежую версию Python с помощью *Homebrew* — инструмента, используемого для управления и установки сторонних пакетов в среде macOS.

Во-первых, в вашем терминале нам нужно убедиться, что установлены инструменты командной строки Apple:

```
$ xcode-select --install
```

Вам нужно будет установить *Homebrew*, установщик пакетов для среды macOS. Чтобы установить его, следуйте инструкциям на https://brew.sh/, и убедитесь, что Homebrew установлен правильно.

После того, как вы установили Homebrew, давайте установим последнюю версию Python:

```
$ brew install python
```

После завершения установки убедитесь в том, что Python установлен правильно:

```
$ python --version

Python 3.8.3
```

Установка в срде Linux

Если вы используете версию Linux на основе Debian, вы можете установить Python 3.8 с помощью apt (или любого другого менеджера пакетов):

```
$ sudo apt-get update
$ sudo apt-get install python3.8
```

После завершения установки убедитесь в том, что Python установлен правильно:

```
$ python --version
Python 3.8.3
```

Если вы не используете дистрибутив Linux на основе Debian, вы можете скомпилировать Python из исходного кода: www.python.org/downloads/source/.

Как запускаются программы Python

Когда вы устанавливаете Python, вы фактически устанавливаете *интерпретатор* — программу, которая переводит написанный код Python в инструкции, которые ваш компьютер понимает и выполняет. Установленный вами интерпретатор называется CPython, это популярный интерпретатор, написанный на языке C.

Вы запускаете программу Python, передавая ее интерпретатору Python в своем терминале:

```
$ python my_program.py
```

Это преобразует ваш код в «компьютерные инструкции» и компьютер выполняет их.

Как ваша ОС знает, где находится интерпретатор Python?

Ваша операционная система имеет общесистемную переменную под названием PATH, содержащую список путей к файлам, которые нужно пройти при поиске программ. Вы можете проверить, что Python установлен, запустив `echo $PATH` в своем терминале. Интерпретатор Python находится в `/usr/local/bin/`. Это проверяется вызовом `which python`.

Управление внешними ресурсами проекта

Каждый проект, который вы создаете, скорее всего, будет использовать внешние библиотеки. Этими внешними ресурсами могут быть библиотеки доступа к базе данных или инструменты, необходимые для анализа документов или веб-сайтов, но важным является то, что они включены в ваш проект.

Управление внешними ресурсами проекта может быть непростой задачей, так как разные внешние ресурсы имеют разные требования — некоторые внешние ресурсы требуют определенных версий Python, другие могут зависеть от родственных внешних ресурсов. Современные проекты Python используют менеджеры пакетов для решения сложных задач по загрузке, установке и обновлению внешних ресурсов. Таким образом, использование менеджера пакетов упрощает вашу жизнь.

Poetry — один из немногих менеджеров внешних ресурсов для Python. Есть и другие, более популярные, например Pipenv. Но после интенсивного использования обоих я обнаружил, что Poetry имеет более понятный интерфейс и более прагматичен в своих целях.

Установка Poetry

Рекомендуемый способ установки Poetry — запуск в терминале следующим образом:

```
$ curl -sSL https://raw.githubusercontent.com/sdispater/poetry/
master/get-poetry.py |
python
```

Если у вас возникнут какие-либо проблемы, обратитесь к официальной документации и инструкциям по установке на сайте Poetry: https://poetry. eustace.io/docs/

Убедитесь в том, что Poetry была установлена правильно, вызвав ее в оболочке:

```
$ poetry

Poetry version 1.0.9

USAGE
  poetry [-h] [-q] [-v [<...>]] [-V] [--ansi] [--no-ansi] [-n]
  <command> [<arg1>] ... [<argN>]

ARGUMENTS
  <command>                 Команда для выполнения
  <arg>                     Аргументы команды

GLOBAL OPTIONS
  -h (--help)               Выводит это справочное сообщение
  -q (--quiet)              Не дает сообщениям появляться
  -v (--verbose)            Увеличивает детализацию сообщений:
                            "-v" обычный вывод, "-vv" более
                            подробного вывод и "-vvv" отладка
  -V (--version)            Отображает версию этого приложения
  --ansi                    Принудительный вывод ANSI
  --no-ansi                 Запрещает вывод ANSI
  -n (--no-interaction)     Запрет интерактивных вопросов

AVAILABLE COMMANDS
  about                     Показывает информацию о Poetry.
  add                       Добавляет новый внешний ресурс в
                            pyproject.toml.
  build                     Создает пакет, по умолчанию как тарбол
                            и колесо.
  Cache                     Взаимодействует скэшем Poetry.
  check                     Проверяет валидность файла
                            pyproject.toml
  config                    Управляет настройками конфигурации.
```

debug	Отладка различных элементов Poetry.
env	Взаимодействует со средой проекта Poetry.
export	Экспорт файла блокировки в альтернативные форматы.
help	Отображает руководство команды.
init	Создает базовый файл pyproject.toml в текущем каталоге.
install	Устанавливает внешние ресурсы проекта.
lock	Блокирует внешние ресурсы проекта.
new	Создает новый проект Python в <path>.
publish	Публикует пакет в удаленном репозитории.
remove	Удаляет пакет из внешних ресурсов проекта.
run	Запускает команду в соответствующей среде.
search	Ищет пакеты в удаленных репозиториях.
self	Взаимодействует с Poetry напрямую.
shell	Создает оболочку в виртуальной среде.
show	Показывает информацию о пакетах.
update	Обновляет внешние ресурсы в соответствии с файлом pyproject.toml.
version	Показывает версию проекта или повышает ее, если указано допустимое правило изменения.

Создание проекта Python с использованием Poetry

Давайте создадим новый проект с использованием Poetry:

```
poetry new my-project
```

Это создает следующую «стандартную» структуру проекта Python в папке с именем my-project:

```
.
├── README.rst
├── my_project
│   └── __init__.py
├── pyproject.toml
└── tests
    ├── __init__.py
    └── test_my_project.py
```

Самый важный файл здесь — это pyproject.toml, который описывает проект и его внешние ресурсы. Он написан в формате разметки под названием TOML:

```
[tool.poetry]
name = "my-project"
version = "0.1.0"
description = ""
authors = ["Daniel van Flymen <vanflymen@gmail.com>"]

[tool.poetry.dependencies]
python = "^3.8"

[tool.poetry.dev-dependencies]
pytest = "^5.2"

[build-system]
requires = ["poetry>=0.12"]
```

```
build-backend = "poetry.masonry.api"
```

Обратите вниманиена то, что единственный внешний ресурс, который у нас есть прямо сейчас, — это Python ^3.8, что указывает на то, что вашему проекту требуется *любая* версия Python до 4.0.0. Poetry также добавила pytest в качестве внешнего ресурса от разработчиков для запуска тестов, но об этом позже.

Установка внешних ресурсов

Одной из самых популярных библиотек Python является библиотека запросов, которая позволяет нам выполнять простые HTTP-запросы. Давайте установим ее:

```
$ poetry add requests

Creating virtualenv my-project-py3.8 in /Users/dvf/Library/
Caches/pypoetry/virtualenvs Using version ^2.22 for requests

Updating dependencies
Resolving dependencies... (0.7s)

Writing lock file

Package operations: 14 installs, 0 updates, 0 removals

  - Installing more-itertools (7.2.0)
  - Installing zipp (0.6.0)
  - Installing importlib-metadata (0.23)
  - Installing atomicwrites (1.3.0)
  - Installing attrs (19.3.0)
  - Installing certifi (2019.9.11)
  - Installing chardet (3.0.4)
  - Installing idna (2.8)
  - Installing pluggy (0.13.0)
  - Installing py (1.8.0)
  - Installing six (1.12.0)
```

```
- Installing urllib3 (1.25.6)
- Installing pytest (3.10.1)
- Installing requests (2.22.0)
```

Используя синтаксис Poetry добавляет запросы ведущие к возникновению ряда вещей:

1. Создана «виртуальная среда» (virtualenv) в /Users/dvf/Library/Caches/pypoetry/ virtualenvs/my-project-py3.8. Virtualenv — это папка, содержащая интерпретатор Python и пакеты вашего проекта, чтобы не засорять остальную часть вашей системы.
2. Последняя версия запросов (2.22.0) была загружена из индекса пакетов Python по адресу https://pypi.org.
3. Все внешние ресурсы, от которых зависят запросы, также были загружены. Это так называемые субресурсы.
4. Все загруженные пакеты были установлены в виртуальной среде вашего проекта.

Теперь мы готовы начать создание и взаимодействие с нашими внешними ресурсами.

Активация virtualenv

Прежде чем вы сможете запустить свой проект, вам необходимо активировать файл virtualenv. То есть необходимо сказать своему компьютеру, что необходимо использовать интерпретатор Python, расположенный в виртуальной среде, которой управляет Poetry.

Во-первых, давайте проверим, какой интерпретатор Python мы используем:

```
$ which python

/usr/local/bin/python
```

Давайте активируем virtualenv:

```
$ poetry shell

Spawning shell within /Users/dvf/Library/Caches/pypoetry/
virtualenvs/my-project-py3.8
```

А теперь давайте проверим, что интерпретатор Python, который мы используем, является интерпретатором виртуальной среды Poetry:

```
$ which python

/Users/dvf/Library/Caches/pypoetry/virtualenvs/my-project-
py3.8/bin/python
```

Проблемы?

Если вы столкнулись с проблемой, вам следует позаботиться о закрытии и повторном открытии терминала и тщательно следовать инструкциям, приведенным в начале этой главы. если вы все еще не решили проблему, то, вероятно, она связана с вашей переменной PATH.

Я предлагаю внимательно прочитать документацию Poetry, чтобы убедиться, что вы выполнили все шаги, так как вполне вероятно, что программное обеспечение могло претерпеть измение с момента публикации этой книги.

Пример: Получение цены биткойна

Давайте начнем с создания кода, который получает текущую спотовую цену биткойна с биржи.

Сначала создайте файл с именем current_price.py в папке my-project/my_project.

Ваша структура папок должна выглядеть следующим образом:

```
.
├── README.rst
├── my_project
│   ├── __init__.py
│   └── current_price.py
├── pyproject.toml
└── tests
    ├── __init__.py
    └── test_my_project.py
```

Мы будем использовать только что установленную библиотеку запросов, чтобы получить цену в долларах США. Введите следующий код в current_price.py:

```
import requests

response = requests.get("https://api.coinbase.com/v2/prices/
spot?currency=USD")
print(response.text)
```

Хорошо, сохраните файл. Давайте запустим его:

```
$ poetry run python my_project/current_price.py

{"data":{"base":"BTC","currency":"USD","amount":"9161.25"}}
```

В предыдущей команде обратите внимание на строку poetry run.... Это ярлык для включения virtualenv и запуска остальной части команды внутри него. Точно так же вы можете включить virtualenv с помощью оболочки Poetry, а затем запустить команду.

Если вы внимательно изучите выходные данные, вы увидите цену $9161,25.

Резюме

В результате мы рассмотрели

1. Установку свежего интерпретатора Python 3.8 на вашу машину
2. Использование Poetry для создания и установки внешних ресурсов для нового проекта (что мы начнем делать в каждой следующей главе)
3. Получение последней цены биткойнов путем вызова API с использованием библиотеки запросов

Теперь, когда вы настроили структуру проекта, мы готовы погрузиться и начать учиться путем практического программирования.

ГЛАВА 2

Способ все идентифицировать

Если вас интересуют блокчейны и криптовалюты, то вы, вероятно, слышали о хэшировании (или хешировании). Идея хеширования является краеугольным камнем криптографии — важнейшим компонентом инфраструктуры блокчейна. В этой главе вы получите практические знания о хешировании и о том, почему это так важно.

Настройка проекта

В духе обучения на практике мы создадим новый проект, который будем развивать постепенно, глава за главой. Мы назовем его funcoin:

```
$ poetry new funcoin
```

Перейдите в папку funcoin:

```
$ cd funcoin
```

Папка funcoin должна выглядеть так:

```
.
├── funcoin
│   └── __init__.py
├── pyproject.toml
└── tests
    ├── __init__.py
    └── test_funcoin.py
```

Структура проекта Эта структура папок приблизительно соответствует тому, как выглядят все современные проекты Python. Она была определена в PEP (Python Enhancement Proposal) 518. Poetry применяет эту структуру папок при создании нового проекта.

Давайте создадим файл для изучения этой главы в *папке проекта* funcoin, назовите его chapter_2.py. Папка вашего проекта теперь должна выглядеть так:

```
.
├── README.rst
├── funcoin
│   ├── __init__.py
│   └── chapter_2.py
├── pyproject.toml
└── tests
    ├── __init__.py
    └── test_funcoin.py
```

Хэш-функции

Теоретически **хеширование** — это акт идентификации данных. Это особый способ присвоения уникального случайного значения любым данным — предложению, фотографии, электронной таблице или загруженной программе. Вы можете думать о хеш-функциях как об «машинах идентификации» — о чем-то, что присваивает значение определенным данным на входе. Данные на входе могут быть любыми — это могут быть изображения, документы, файлы, необработанные байты, числа, что угодно, — но **результат на выходе всегда предсказуем для одних и тех же данных на входе**.

Соглашения об именах Люди склонны использовать термин **хэш** и как глагол, и как существительное: «Проверьте хэш этой фотографии» (существительное) или «Давайте хешируем следующий файл» (глагол). Правильный термин для данных на выходе хеш-функции — «дайджест», но «хэш» стал обычным явлением.

УникальностьВо втором предложении выше утверждается, что хэши уникальны и случайны — это немного не так. В этой книге мы имеем дело с особым классом хэш-функций, называемых криптографическими хеш-функциями, в которых вычисляемый хеш выбирается из достаточно большого пула значений, так что две величины на входе, имеющие одно и то же значение, крайне маловероятны в известной нам вселенной. Итак, перефразируя предложение: *хэши настолько хороши, насколько вы можете на рактике достичь уникальности*.

Пример 1: Хэширование в Python

Python поставляется со стандартным набором популярных хеш-функций; они доступны в библиотеке hashlib. Давайте запустим Python и начнем практиковаться с ними.

Листинг 2-1. hashing_strings.py

```
import hashlib

# Hash functions expect bytes as input: the encode() method
turns strings to bytes
input_bytes = b"backpack"

output = hashlib.sha256(input_bytes)

# We use hexdigest() to convert bytes to hex because it's
easier to read
print(output.hexdigest())
```

Какой результат вы получили? Это должно было быть:

```
5f00368a6ad231c3c439c4f6bc33c27014b4d35a904ff1656d74f9528636f496.
```

Теперь попробуйте изменить одну из букв в слове "backpack" на заглавную. Попробуйте добавить пробел в конце. Играйте с вводом каждый раз. Вы заметили, что полученные хэши *совершенно* разные? Хорошо. В таком случае вы только что открыли эффект *лавины*: **незначительные изменения данных на входе приводят к большим изменениям результата на выходе**.

Как правило, хеш-функции считаются **криптографическими**, если они удовлетворяют следующим свойствам:

- **Детерминированная:** Одини и те же данные на входе всегда дают один и тот же хэш.
- **Стойкость:** Невозможно вычислить данные на входе для заданного хэша, кроме как путем прямого перебора (испытание огромного количества возможных данных на входе).
- **Безопасность при коллизиях:** Невозможно найти две разные пары данных на входе, которые на выходе дают один и тот же хеш.

- **Лавинный эффект:** Самое незначительное изменение в данных на входе должно давать новый хеш, настолько отличающийся от правильного, что новый хеш должен абсолютно не коррелировать со старым хэшем.
- **Скорость:** Хэш должен генерироваться автоматически и *быстро*.

Выбор хеш-функции Одноранговые блокчейны делают свой выбор хэш-функции известным в своих протоколах: Биткойн использует двойной sha256, в то время как Ethereum использует keccak256. Важно знать, что все эти хеш-функции делают одно и то же: они обеспечивают предсказуемый результат на выходе для заданных данных на входе.

Давайте подведем итог того, что мы обнаружили на данный момент.

Трмин	Объяснение	Пример
hash function	Функция, которая идентифицирует все данные, что вы вводите, выводя гигантское случайное шестнадцатеричное число. Это число *всегда* одинаково для одних и тех же данных на входе.	Криптографическая хэш-функция, такая как sha256, md5, blake2b, и т.д.
hash/ digest	Данные на выходе хеш-функции: огромное случайное шестнадцатеричное число.	Хэш для слова “dan”: ec4f...f1cb

Пример 2: Хеширование изображений

Вот изображение купюры в 100 долларов.

Рис. 2-1. *Купюра в* 100 *долларов*

Хэш этого изображения e25641dc52387baba19751783ae4e060.

Вот то же изображение, но с небольшим изменением.

Рис. 2-2. *Купюра в* 100 *долларов (изменнная)*

Можете ли вы найти изменение? Может быть да, а может и нет. Присмотрись (Я изобразил слово "STATES" без буквы S). Но вот что интересно; если я пропускаю это изображение через хеш-функцию, то получаю результат f4c56f530133b8de6b3b0b39a610be32. Это не может удивить, но если вы сравните два хэша, вы заметите, что они сильно различаются и кажутся случайными. Я сравню их для вас:

e25641dc52387baba19751783ae4e060
f4c56f530133b8de6b3b0b39a610be32

Пример Прорывная идея, лежащая в основе Биткойна, заключается в том, что хэши могут использоваться для доказательства того, что компьютером была выполнена работа — мы будем исследовать это в следующих главах, но пока имейте в виду, что Биткойн использует кажущуюся «случайность» криптографических хэшей.

Давайте попробуем хешировать наше собственное изображение (или файл) в Python.

Листинг 2-2. hashing_files.py

```
from hashlib import sha256

file = open("my_image.jpg", "rb")
hash = sha256(file.read()).hexdigest()
file.close()

print(f"The hash of my file is: {hash}")
```

Давайте рассмотрим предыдущий код и объясним его, строка за строкой:

```
from hashlib import sha256
```

Эта строка импортирует хеш-функцию sha256 из библиотеки hashlib, входящей в состав Python. sha256 - одна из самых

популярных хеш-функций (она широко используется в биткойнах).

Здесь мы открываем файл с именем my_image. jpg, находящийся в том же каталоге, что и ваш код:

```
file = open("my_image. jpg", "rb")
```

Аргумент "rb" означает, что файл должен быть в режиме только для чтения и читаться как байты.

```
hash = sha256(file. read()). hexdigest()
```

Затем мы читаем открытый файл нашей хеш-функцией, например: sha256(file. read()), и присваиваем hexdigest(), то есть данные на выходе в виде шестнадцатеричной строки, переменной с именем hash.

Наконец, мы закрываем открытый файл и выводим хэш на терминал.

Проверка файлов, загруженных из сети Интернет Вы можете использовать приведенный выше сценарий, чтобы убедиться, что загруженный файл не был изменен третьей стороной. Авторитетные веб-сайты информируют, каким должен быть хэш (иногда называемый контрольной суммой) файла, чтобы вы могли проверить его локально.

Аналогии

Если вы все еще в неведении, вот аналогия из реальной жизни, чтобы донести идею до вашего сознания.

Представьте, что вы в аэропорту, проходите контроль безопасности и бросаете свой рюкзак на конвейерную ленту. Но конвейерная лента оснащена специальным сканером — машиной, которая делает рентгеновский снимок ваших вещей и выдает хеш-код.

Рюкзак сканируется, и сканер выдает 8-значный хэш, идентифицирующий ваш рюкзак: 13371817. Если ваш рюкзак и

все предметы в нем точно такие же, сканер всегда будет выдавать один и тот же номер. Но если вы сейчас решите достать предмет из рюкзака, сканер выдаст совсем другой номер.

Необратимость

Как мы уже говорили, в блокчейнах используются **криптографические** хэш-функции, которые чрезвычайно сложно обратить вспять. Это означает, что если нам дан хеш, невероятно сложно (даже на уровне суперкомпьютеров) когда-либо угадать, какие данные были на входе.

В предыдущем примере, если бы я дал вам хэш e25641dc52387baba19751783ae4e060, вам было бы очень трудно догадаться, что это была фотография 100-долларовой банкноты.

Крайне важно понять эту концепцию, поскольку она составляет основу того, что обеспечивает безопасность биткойнов и блокчейнов в целом. И, как мы увидим позже, существует много разных хеш-функций с разными свойствами.

Пример 3: Отправка незащищенных электронных писем

Проблема

Алиса хочет отправить Бобу электронное письмо по незащищенному каналу, такому как сеть Интернет. Боба не интересует, могут ли другие люди прочитать электронное письмо, но он хочет убедиться, что оно не было подделано. Как Боб может убедиться, что сообщение электронной почты не было подделано?

Примечание Прежде чем мы сформулируем решение, используя знания о хэшировании, которые вы только что усвоили, не могли бы вы подумать, как можно разработать такую схему?

Решение

Общая идея состоит в том, чтобы Алиса и Боб заранее договорились об общем секрете. Затем Алиса удаляет секретную фразу из сообщения и отправляет ее Бобу вместе с хешем. Затем Боб выполняет ту же процедуру, что и Алиса: он добавляет предварительно сообщенную секретную фразу и выводит хэш. Если хэш Боба не совпадает с тем, что отправила Алиса, то он понимает, что сообщение было подделано.

Подведем итоги

1. Алиса и Боб используют секретную фразу S.
2. Затем Алиса создает хэш H сообщения M с добавлением секрета в конец сообщения: H = hash(M + S).
3. Алиса отправляет H и M Бобу (сообщение и вычисленный хэш).
4. Боб проверяет целостность сообщения, самостоятельно вычисляя H, чтобы увидеть, совпадает ли оно с хэшем, присланным Алисой.

Примечание Эта общая схема описывает ряд протоколов, называемых hMaC (хешированный код аутентификации сообщений), и описана в IETf rfC 2104.

Давайте посмотрим на пример. Прежде чем Алиса и Боб начнут общение, они оба согласовывают секретный пароль, *bolognese*. Алиса создает свое электронное письмо по заданной схеме. Вот как оно выглядит, когда Боб получает его:

От: <Alice> alice@example.com

Тема: Ты слышал о биткойнах?

Эй, Боб, я думаю, тебе стоит узнать о блокчейнах! Я

инвестировала в Биткойн, и в настоящее время на моем счете ровно 12,03 BTC.

хэш: 71890dc61c21370874d2a7b74064396cb613a1924f09aa06925abc7842e6802c

Боб замечает, что Алиса включила хеш в тело письма. Он удаляет хэш, затем добавляет секретную фразу (*bolognese*) к телу письма и вычисляет его хэш. Если хэш совпадает с включенным в тело письма хешем, то Боб может сделать вывод, что сообщение не было подделано (а у Алисы действительно есть 12,03 BTC).

Важно, чтобы и Алиса, и Боб знали протокол, который они будут использовать для проверки. Он включает в себя используемую хеш-функцию (sha256) и место для размещения секретной фразы в контексте сообщения (они соглашаются добавить ее в конец), а также любые другие данные, которые могут дополнительно защитить схему (например, метку времени).

Давайте посмотрим, как Боб может проверить сообщение Алисы в Python..

Листинг 2-3. hashing_emails.py

```
from hashlib import sha256

secret_phrase = "bolognese"

def get_hash_with_secret_phrase(input_data, secret_phrase):
    combined = input_data + secret_phrase
    return sha256(combined.encode()).hexdigest()

email_body = "Эй, Боб, я думаю, тебе стоит узнать о
блокчейнах! " \
             "Я инвестировала в Биткойн, и в настоящее время на
             моем счете ровно 12,03 BTC."

print(get_hash_with_secret_phrase(email_body, secret_phrase))
```

Когда Боб запустит этот код, он сможет проверить хэш Алисы.

Как предотвращение спама привело к доказательствам работы

Знаете ли вы, что хеш-функции можно использовать для предотвращения спама в электронной почте? Алгоритм, лежащий в основе этой идеи, называется алгоритмом доказательства работы или «Proof-of-Work», и он составляет основу Биткойна. Алгоритм был изобретен британским криптографом Адамом еще в 1997 году, он назвал его Hashcash.

Общая идея состоит в том, чтобы принимать только те электронные письма, хэши которых удовлетворяют ограничению. Это заставляет отправителя электронной почты выполнять некоторую вычислительную работу перед отправкой электронной почты. Другими словами, отправителю становится накладно рассылать спам людям.

Но что такое ограничение? Хэши — это просто числа, но они кажутся случайными, поэтому мы можем применить любое ограничение, которое захотим. Я мог бы установить правило, согласно которому каждый, кто отправляет мне электронное письмо, должен предоставить хеш электронного сообщения, который является нечетным числом. Но это не было бы хорошим ограничением, поскольку отправитель генерировал бы 50% в виде нечетных хэшей.

Алгоритм Hashcash применяет ограничение, согласно которому значение хеш-функции должно быть меньше определенного числа. Это звучит обманчиво просто, но подумайте об этом на мгновение: поскольку криптографические хеш-функции выдают случайное число для заданных данных на входе, как мы можем гарантировать, что тело письма хешируется до определенного числа? Мы этого сделать не можем. Единственное, что мы можем сделать, это вставить произвольные данные в тело письма и продолжать методом проб и ошибок, пока наш хеш не будет

удовлетворять ограничению.

Давайте рассмотрим пример, а также наложим ограничение, согласно которому тело электронного письма должно иметь число меньше 10 000.

Примечание Я использую восьмизначную хеш-функцию, которая для простоты выводит только десятичные числа.

Листинг 2-4. Должно ли это письмо быть принято нашим сервером?

```
От: dan@example.com

Тема: Покупай биткойны!

Эй, Итан, я думаю, тебе стоит купить биткойны!

хэш: 95119035
```

Нет. Хэш тела этого письма — 95119035, то есть он больше 10 000. Таким образом, отправитель не выполнил требуемой работы: он должен убедиться, что хэш тела письма удовлетворяет ограничению.

Единственный способ, которым отправитель может сделать это, вставить произвольные данные в тело письма:

```
От: dan@example.com

Тема: Покупай биткойны!

Эй, Итан, я думаю, тебе стоит купить биткойны!

s8763ASdh727212213098

хэш: 00000891
```

Теперь мы видим, что хэш тела равен 891, что удовлетворяет нашему ограничению! (Я дополнил хеш нулями, потому что шестнадцатеричные числа обычно отображаются именно так). Наш почтовый сервер хеширует тело письма и видит, что это действительно 891, и поэтому принимает письмо.

Вот что такое доказательство работы: работа подтверждается отображением определенного хэша, и сервер может легко это проверить. Другой способ думать об этой схеме состоит в том, что ее трудно сгенерировать, но легко проверить.

Примечание Это важная идея, которая позволяет любому блокчейну с доказательством работы генерировать (или добывать) деньги, однозначно доказывая, что вычислительная работа была выполнена для всех в сети. При этом энергия расходуется на использование кажущейся случайной природы криптографических хэшей!

Резюме

К этому моменту вы должны были познакомиться с функциями хэшированияи и ***хешированием*** произвольных данных. Подводя итог, вам должно быть комфортно

1. Импоровать библиотеки hashlib в Python
2. Хешировать в Python любые структуры данных или бинарный файл
3. Понимать то, что криптографические хеш-функции невероятно сложно обратить

Все равно не поняли сути?

Это нормально. Это тяжело. Если вы все еще не уверены в своих знаниях хэширования, стоит остановиться здесь и просмотреть эту главу еще раз, пока она не впитается. Возможно, это самый важный аспект структуры данных блокчейна, и мы будем широко использовать его в будущем.

ГЛАВА 3

Блокчейны

В этой главе мы с головой погрузимся в блокчейн, используя простые типы данных в Python. Вы закончите эту главу с фундаментальным пониманием того, что такое блокчейн, что у него внутри и как хэши используются для обеспечения его устойчивости.

Концепции предыдущей главы имеют решающее значение для понимания того, почему блокчейны считаются неизменяемыми (читай, неподдающимися изменению). Мы уже видели, как отправлять незащищенные электронные письма. Мы расширим эти идеи, чтобы построить блокчейн с нуля, и с четким пониманием задействованных структур данных покажем, почему мошенничество в блокчейне невозможно.

Давайте погрузимся в блоки.

Как выглядит блок?

Наши блоки — это простые словари Python. Вот пример того, как может выглядеть отдельный блок в Python:

```
block_1038 = {
    'index': 1038,
    'timestamp': "2020-02-25T08:07:42.170675",
    'data': [
        {
            'sender': "bob",
```

```
            'recipient': "alice",
            'amount': "$5",
        }
    ],
    'hash': "83b2ac5b",
    'previous_hash': "2cf24ba5f"
}
```

Каждый блок имеет предыдущую базовую структуру, но я бы хотел, чтобы вы сосредоточились на двух последних полях: хеш (hash) и предыдущий хэш (previous_hash). Каждый блок содержит внутри себя хэш предыдущего блока. Блок 1038 содержит хеш блока 1037, который, в свою очередь, содержит хеш блока 1036, и так далее… назад к первому блоку — блоку «генезиса».

Блок может содержать *любые* данные: файлы, изображения, транзакции, записи и т.д.

В предыдущем примере наш блок содержит одну транзакцию от Боба к Алисе за 5 долларов. Этот блок похож на то, как выглядит большинство блоков криптовалюты (например, биткойн). Возможно, вы слышали, что люди описывают Ethereum как «мировой компьютер». Это связано с тем, что блоки Ethereum также содержат *исполняемый код* как часть своих данных, инструктируя участников сети выполнять операции с самой цепочкой блоков.

Неизменяемость и важность хэшей

Идея цепочки не должна быть слишком растяжимой — **каждый блок содержит в себе хэш предыдущего блока, образуя цепочку**. Это «связывание» хэшей — это то, что придает блокчейнам их свойства неизменности и предотвращения мошенничества.

В частности, поле previous_hash - это ссылка между блоками для создания *цепочки*. Если злоумышленник каким-то образом изменили более ранний блок в цепочке, то все последующие блоки будут изменены, а их хэши будут неверными. Например, если бы нам нужно было изменить один фрагмент данных в блоке № 1037, то хэш блока № 1037 стал бы другим, а потому значение

previous_hash в блоке № 1038 было бы неверным. **Таким образом, если хоть один бит в любом более раннем блоке будет подделан, вся последующая цепочка будет недействительной.** Это цепная природа блокчейнов — они защищены цепочкой хэшей с использованием previous_hash:

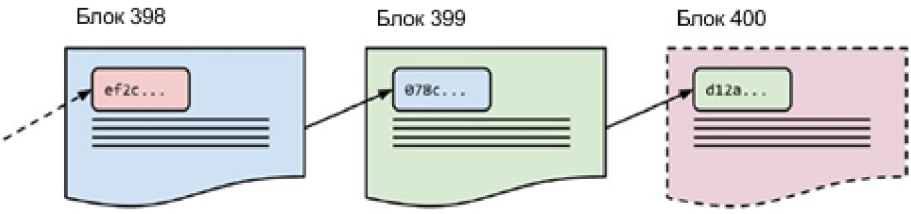


Рис. 3-1. *Хэши блоков, образующих цепочку*

Это «особый соус» Биткойна — цемент, который держит все вместе. Если бы кто-то обманул блок, изменив сумму транзакции где-то в цепочке, поле previous_hash следующего блока было бы другим, что привело бы к изменению хэшей. И все участники сети Биткойн сразу же заметили бы несоответствие и проигнорировали бы это изменение!

Базовый блокчейн на Python

Откройте ваш любимый текстовый редактор или IDE (лично я использую PyCharm) и создайте новый файл с именем blockchain.py. Пока мы будем использовать только один файл, но если вы заблудитесь, вы всегда можете обратиться к исходному коду по адресу https://github.com/dvf/blockchain.

Представление блокчейна с помощью класса

Для начала мы создадим простой класс блокчейна. К концу книги этот класс будет более продвинутым, а пока мы собираемся осторожно представить различные концепции, добавив в наш класс отдельные методы.

Давайте начнем с конструктора некоторых методов для создания проекта:

Листинг 3-1. blockchain.py

```
class Blockchain(object):
    def __init__ (self):
        self.chain = []

    def new_block(self):
        # Генерирует новый блок и добавляет его в цепь

    @staticmethod
    def hash(block):
        # Hashes a Block
        pass

    def last_block(self):
        # Получает последний блок в проходе
        цепочки
```

В нашем проекте есть конструктор, который создает начальную пустую цепочку списков (для хранения нашей цепочки блоков) с дополнительными методами new_block, create_block, hash, и last_block для создания новых блоков, их хеширования и получения последних.

Здесь идея в том, что метод new_block отвечает за создание блоков и добавление их в цепочку. Наш класс Blockchain будет отвечать за все операции, необходимые для поддержания нашей цепочки блоков. Давайте начнем детализировать некоторые из этих методов.

Сопровождение транзакций

Далее в книге транзакции занимают целую главу. Они включают в себя кучу знаний по криптографии и цифровым подписям.

Начнем с того, что наш блокчейн будет поддерживать примитивные неподписанные транзакции — просто для иллюстрации — поэтому мы создадим метод с именем new_transaction:

Листинг 3-2. blockchain.py

```
class Blockchain(object):
    def __init__ (self):
        self.chain = []
        self.pending_transactions = []

    def new_block(self):
        # Генерирует новый блок и добавляет его в цепь

    @staticmethod
    def hash(block):
        # Хэширует блок

    def last_block(self):
        # Возвращает последний блок в цепочке

    def new_transaction(self, sender, recipient, amount):
        # Adds a new transaction to the list of pending
        transactions
        self.pending_transactions.append({
            "recipient": recipient,
            "sender": sender,
            "amount": amount,
        })
```

Но пока мы сохраним наши транзакции простыми. Позже, в ГЛАВЕ 6, мы узнаем больше о криптографии, лежащей в основе транзакций, и о том, как они поддерживаются в производственных блокчейнах, таких как Биткойн.

Добавление блоков

Когда наша цепочка блоков будет создана, нам нужно будет наполнить ее блоком генезиса — блоком, не имеющим предшественников и с индексом 0. Это особый случай блока, который почти всегда жестко запрограммирован в программном обеспечении. В Биткойне блок генезиса был создан Сатоши Накамото и, как известно, содержал следующий текст (в шестнадцатеричном формате) из статьи первой полосы газеты *The Times* от 03 января 2009 года:

The Times 03/Jan/2009 Chancellor on brink of second bailout for banks

Теперь мы начнем добавлять некоторые функции; итак..

- Конкретизируем метод `new_block()`.
- Конкретизируем метод `hash()` (как и в биткойнах, для хэширования мы будем использовать хеш-функцию SHA-256).
- Добавим блок генезиса в метод конструктора.

```
1  import json
2
3  from datetime import datetime
4  from hashlib import sha256
5
6
7  class Blockchain(object):
8      def __init__ (self):
9          self.chain = []
10         self.pending_transactions = []
11
12         # Создает блок генезиса
13         print("Creating genesis block")
14         self.new_block()
```

```
15
16      def last_block(self):
17          # Возвращает последний блок в цепочке (если есть
              блоки)
18          return self.chain[-1] if self.chain else None
19
20      def new_block(self, previous_hash=None):
21          block = {
22              'index': len(self.chain),
23              'timestamp': datetime.utcnow().isoformat(),
24              'transactions': self.pending_transactions,
25              'previous_hash': previous_hash,
26          }
27          # Возвращает хэш этого нового блока и добавляет его в
            блок
28          block_hash = self.hash(block)
29          block["hash"] = block_hash
30
31          # Сбрасывает список незавершенных транзакций
32          self.pending_transactions = []
33          # Добавляет блок в цепочку
34          self.chain.append(block)
35
36          print(f"Created block {block['index']}")
37          return block
38
39          @staticmethod
40          def hash(block):
41              # Гарантирует, что словарь отсортирован, иначе у нас
                будут несогласованные хэши
42              block_string = json.dumps(block, sort_keys=True).
                encode()
43              return sha256(block_string).hexdigest()
```

На этом этапе вы можете открыть Python в интерактивном режиме и начать экспериментировать с вашим классом Blockchain:

```
$ poetry shell
$ python -i blockchain.py
```

Создание блокчейна:

```
>>> bc = Blockchain()
Creating genesis  block
Created block 0
```

Мы должны увидеть, что в цепочке есть только один блок — блок генезиса (для краткости я обрезал хэши):

```
>>> bc.chain
[{"index": 0, "timestamp": "2019-02-25T14:23:08.853678",
"transactions": [], "previous_hash": None, "hash":
"80ad...01bd"}]
```

Попробуйте добавить новый блок:

```
>>>  bc.new_block(previous_hash="80ad...01bd")
Created block 1
```

Продолжите экспериментировать с блокчейном.

Полный код blockchain.py

```
1  import json
2
3  from datetime import datetime
4  from hashlib import sha256
5
6
7  class Blockchain(object):
```

```
8       def __init__ (self):
9           self.chain = []
10          self.pending_transactions = []
11
12          # Create the genesis block
13          print("Creating genesis block")
14          self.new_block()
15
16      def new_block(self,  previous_hash=None):
17          block = {
18              'index': len(self.chain),
19              'timestamp': datetime.utcnow().isoformat(),
20              'transactions': self.pending_transactions,
21              'previous_hash':  previous_hash,
22          }
23
24          # Возвращает хэш этого нового блока и добавьте его в
            блок
25          block_hash = self.hash(block)
26          block["hash"] = block_hash
27
28          # Сброс списка незавершенных транзакций
29          self.pending_transactions = []
30          # Добавление блока в цепочку
31          self.chain.append(block)
32
33          print(f"Created block {block['index']}")
34          return block
35
36      @staticmethod
37      def hash(block):
```

```
38          # Мы гарантируем, что словарь отсортирован, иначе у нас
              будут несогласованные хэши
39          block_string = json.dumps(block, sort_keys=True).
            encode()
40          return sha256(block_string).hexdigest()
41
42      def last_block(self):
43          # Возвращает последний блок в цепочке (если есть блоки)
44          return self.chain[-1] if self.chain else None
```

Код должен быть простым. Я добавил несколько комментариев в код, чтобы сделать его понятным. Обратите внимание, что когда создается экземпляр нашего класса Blockchain, мы наполняем его блоком *генезиса* — блоком, у которого нет предшественников.

В этот момент вам должно быть интересно, как люди в сети соглашаются добавлять новые блоки в свои блокчейны. Поскольку мы хотим полностью децентрализованную одноранговую сеть, должен существовать общий набор правил (протокол), которому следуют все участники; это и есть правила игры. Добавление новых блоков в цепочку блоков является результатом майнинга.

Давайте погрузимся в этот вопрос.

ГЛАВА 4

Доказательство работы

Основная цель этой главы — четко объяснить, как добываются блоки в блокчейне. Косвенно это объясняет, как появляются новые монеты криптовалюты, а также то, как разрозненные люди, образующие сеть, могут прийти к консенсусу (договориться о состоянии блокчейна). В результате вы закончите эту главу с практическим пониманием доказательства работы или Proof of Work (PoW) — протокола, который позволяет достичь это.

Мы также увидим, что из-за одноранговой, децентрализованной природы одноранговых сетей в любой момент времени никогда не бывает единой цепочки блоков. Существует много действительных цепочек одновременно, но со временем участники приходят к консенсусу относительно того, что является легитимной цепочкой.

В предыдущей главе мы реализовали методы в нашем классе блокчейна. Мы также узнали, как блоки «соединяются» вместе с помощью хэшей и реализовали методы для создания и добавления новых блоков. Крайне важно, чтобы вы хорошо разбирались в хешировании, так как в этой главе начинается настоящее волшебство — мы разъясним концепцию «майнинга» или, более формально, создадим доказательства работы.

Взаимодействие с классом блокчейна с помощьюiPython

Отличным инструментом, помогающим взаимодействовать с интерпретатором Python, является iPython. Он добавляет в ваш интерпретатор Python автодополнение табуляций и подсветку синтаксиса, что существенно упрощает просмотр и понимание взаимодействия. Установите его, запустив сначала виртуальную среду и используя pip:

```
$ pip install ipython
```

Затем вызовите интерпретатор Python:

```
$ ipython -i funcoin/examples/chapter_3/full_blockchain.py

Python 3.8.3 (default, Oct 13 2019, 18:33:25)
Type 'copyright', 'credits', or 'license' for more information.
IPython 7.12.0 -- An enhanced Interactive Python. Type '?' for
help.

In [1]:
```

Попробуйте вызвать свою цепочку блоков и использовать добавление табуляции в методах:

```
In [1]: bc = Blockchain()
Creating genesis block
Created block 0

In [2]: bc.h
        hash
        chain
        last_block
        new_block
        pending_transactions
```

В будущем вам нужно будет проявлять инициативу,

экспериментируя со своим классом блокчейна, чтобы получить интуитивное понимание будущих концепций.

Введение в PoW

При формировании Биткойна гениальность Сатоши Накамото заключалась не в создании, а скорее в объединении ранее существовавших прорывовных идей криптографов, озабоченных проблемами конфиденциальности. Эти идеи охватывают разрозненные области и темы, такие как решение проблемы централизации и доверия за счет использования одноранговых сетей, первоначально использовавшихся для обмена файлами через Интернет (торренты); или, более конкретно, использование технологии, изобретенной для борьбы со спамом, передаваемым по электронной почте, и для решения проблемы «чеканки» криптомонет.

Но самым большим прорывом Биткойна стало использование доказательств работы для формирования консенсуса среди участников сети, которые не знают друг друга и, что более важно, не доверяют друг другу. Проще говоря: в децентрализованной сети, где все имеют равное право голоса, нам нужен способ согласования решений; например, является ли транзакция мошеннической или легитимна ли цепочка блоков. Теперь, возможно, мы поторопились (скоро мы проясним большинство этих концепций), но важно уменьшить масштаб и посмотреть, куда подходят разные части головоломки.

Доказательство работы или, более формально, алгоритм доказательства работы (PoW) описывает метод, с помощью которого новые блоки добавляются в цепочку блоков. Как Биткойн, так и Ethereum (в настоящее время) используют алгоритмы доказательства работы для добавления новых блоков в свои блокчейны. Алгоритмы PoW кажутся довольно сложными, но на практике они очень просты. *Чтобы заставить шестеренки работать, прежде чем мы*

углубимся, я хотел бы поставить перед вами проблему: как вы докажете кому-то, что вы действительно работали?

Подумайте об этом минуту или две. Допустим, вы копаете ямы на заднем дворе и находите золотой самородок. Является ли этот золотой самородок доказательством того, что для его получения были предприняты усилия? Я бы сказал, что это доказательство того, что ***кто-то*** проделал ***какую-то*** работу. Давайте на секунду сменим тему — как насчет доказательства того, что вы пробежали марафон? Вероятно, вы могли бы прикрепить камеру к голове и записать всю гонку шаг за шагом. Этого было бы достаточно, не так ли? Возможно, но кадры могли бы быть изменены, или камеру мог носить кто-то другой. Возможно, мы тут хватаемся за соломинку, но я пытаюсь обратить ваше внимание на то, насколько сложно доказать, что работа была сделана без сомнений (или доверия), особенно если вы сидите за компьютером, что весьма удобно. при фальсификации.

Проще говоря, алгоритмы доказательства работы предназначены именно для этого — они позволяют вам доказать, что ваш компьютер выполнил какую-то работу. С технической точки зрения, алгоритмы доказательства работы представляют собой механизмы консенсуса: если вы можете доказать, что работа была выполнена, то вы можете показать свое доказательство кому-то еще, кто может легко проверить его истинность. Существует множество вариантов таких алгоритмов, но мы сосредоточимся на простом примере, чтобы понять его.

Мы уже объясняли, что PoW — это алгоритм добычи новых блоков. Фактический метод алгоритма доказательства работы прост - найти число, которое удовлетворяет решению небольшой математической задачи: число должно быть трудно найти, но легко проверить с вычислительной точки зрения любым пользователем сети. Это основная идея доказательства работы: ***сложно сделать, но легко проверить***. Мы рассмотрим очень простой пример, чтобы помочь вам понять это.

Тривиальный пример PoW

Давайте решим, что *хэш* некоторого целого числа x, умноженного на другое целое число y, должен заканчиваться на 0. Поэтому, хэш(x * y) = dedb2ac23dc...0. И для этого упрощенного примера зафиксируем x = 5. Реализация этого на Python:

```
1  from hashlib import sha256
2
3
4  x = 5
5  y = 0  # We don't know what y should be yet...
6
7  while  sha256(f'{x*y}'.encode()).hexdigest()[-1] != "0":
8      y += 1
9
10 print(f'The solution is y = {y}')
```

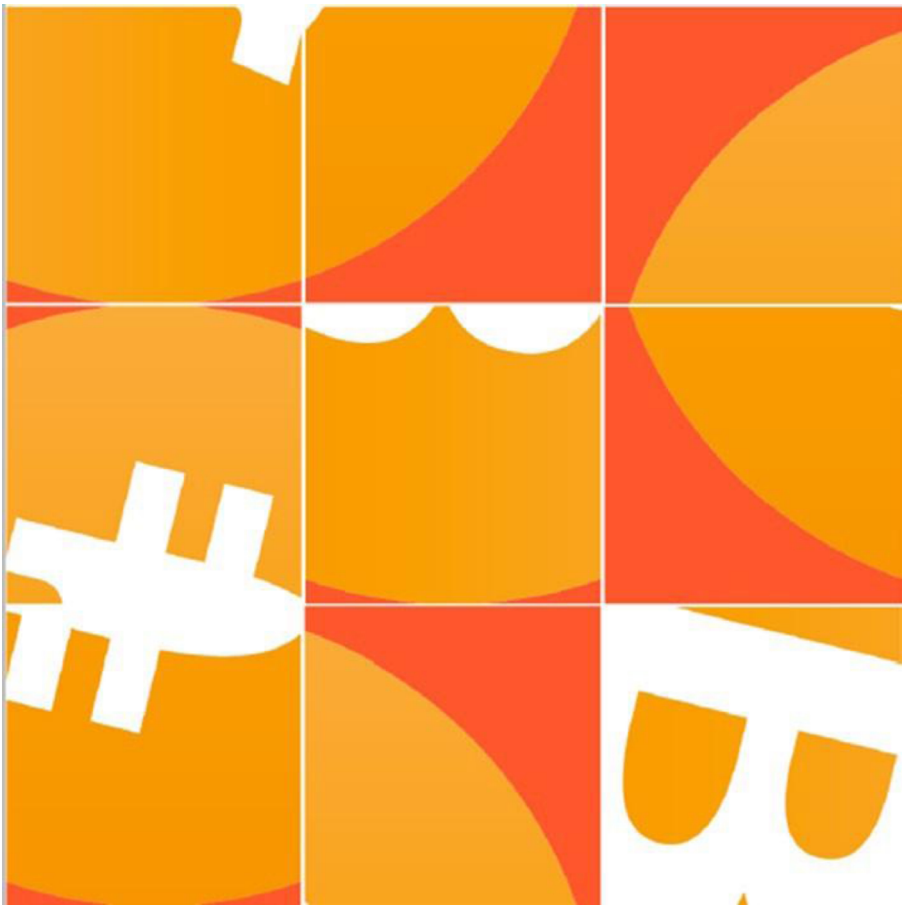
Решение y = 21. Поскольку полученный хеш заканчивается на 0.

```
>>> sha256(f"{5 * 21}".encode()).hexdigest()
'1253e9373e781b7500266caa55150e08e210bc8cd8cc70d89985e
3600155e860'
```

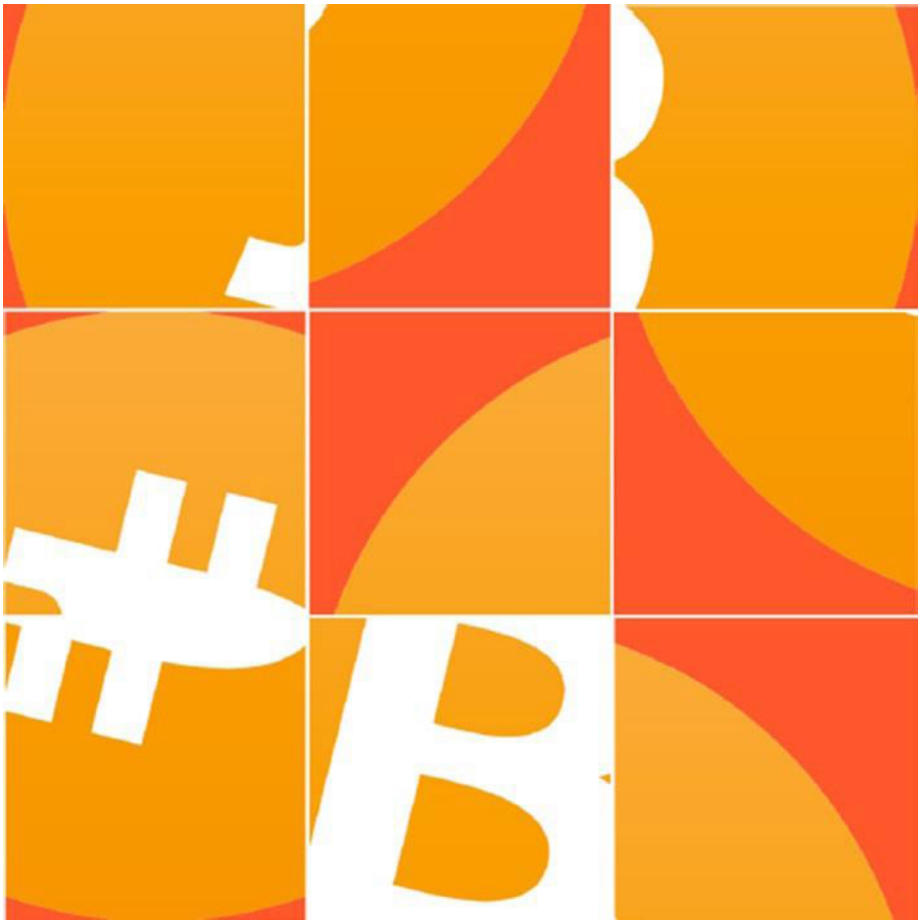
В Биткойне у алгоритма доказательства работы есть имя; его называют *Hashcash*. И это не слишком отличается от предыдущего базового примера. Это алгоритм задачи, которую майнеры пытаются решить, чтобы создать новый блок. Обычно сложность определяется количеством нулей в начале строки; затем майнеры вознаграждаются за свое решение, получая определенное количество биткойнов — за транзакцию; а сеть способна легко *проверить* их решение, так как мы это делали ранее.

Аналогия: пазлы

Головоломки — хороший пример алгоритма доказательства работы, поскольку доказательство работы заключается в единодушной (человеческой) идентификации изображения. Если бы мы попросили компьютер разгадать головоломку



ему пришлось бы рассматривать каждую комбинацию



и это будет продолжаться то тех, пока мы не нажмем «стоп», когда увидим изображение:

И теперь вы могли бы показать это изображение кому-нибудь, чтобы они знали точно, что работа сделана.

Внедрение PoW

Давайте реализуем аналогичный алгоритм для нашего блокчейна. Наше правило будет похоже на предыдущий пример:

Найдите число p, при хешировании которого с помощью решения предыдущего блока получается хэш с четырьмя ведущими нулями.

Продолжая наш пример из предыдущей главы, давайте добавим в наш класс Blockchain два новых метода, proof_of_work и valid_hash, для майнинга и проверки хэшей соответственно:

```
import json

from datetime import datetime
from hashlib import sha256


class Blockchain(object):
    def __init__ (self):
        self.chain = []
        self.pending_transactions = []

        # Создаем блок генезиса
        print("Creating genesis block")
        self.new_block()

    def new_block(self, previous_hash=None):
        block = {
            'index': len(self.chain),
            'timestamp': datetime.utcnow().isoformat(),
            'transactions': self.pending_transactions,
```

```
21              'previous_hash': previous_hash,
22          'nonce': None,23
            }
24
25          # Получаем хэш этого нового блока и добавляем его в
            блок
26          block_hash = self.hash(block)
27          block["hash"] = block_hash
28
29          # Сброс списка незавершенных транзакций
30          self.pending_transactions = []
31          # Добавляем блок в цепочку
32          self.chain.append(block)
33
34          print(f"Created block {block['index']}")
35          return block
36
37      @staticmethod
38      def hash(block):
39          # Мы гарантируем, что словарь отсортирован, иначе у
            нас будут несогласованные хэши
40          block_string = json.dumps(block, sort_keys=True).
            encode()
41          return sha256(block_string).hexdigest()
42
43      def last_block(self):
44          # Возвращает последний блок в цепочке (если есть блоки)
45          return self.chain[-1] if self.chain else None
46
```

```
47      def proof_of_work(self):
48          pass
49
50      def valid_hash(self):
51          pass
```

Мы также добавили в нашу блочную структуру новое поле под названием nonce, что означает ***бессмысленная*** строка. Думайте о nonce как об одноразовом случайном числе, которое будет использоваться как важный источник случайности для наших блоков. Для этих одноразовых случайных чисел мы сгенерируем случайное 64-битное шестнадцатеричное число, используя модуль random Python:

```
import random
>>> format(random.getrandbits(64), "x")
'828ad30173db207b'
```

Давайте конкретизируем наши два метода, proof_of_work и valid_hash, начиная с последнего. valid_hash прост; ему нужно проверить, начинается ли хэш блока с определенного количества нулей, скажем, 4:

```
@staticmethod
def valid_block(block):
    return block["hash"].startswith("0000")
```

Теперь давайте реализуем простой алгоритм PoW, который создает новый блок и проверяет, имеет ли он легитимный хэш:

```
1  def proof_of_work(self):
2      while True:
3          new_block = self.new_block()
4          if self.valid_block(new_block):
5              break
6
7      self.chain.append(new_block)
```

```
8       print("Found a new block: ", new_block)
```

Код говорит сам за себя:

1. Мы создаем новый блок (который содержит случайный одноразовый номер).
2. Затем хэшируется блок, чтобы убедиться, что он действителен.
3. Если он легитимный, то возвращаем его; в противном случае каждый раз повторяем с 1.

Давайте добавим эти методы в наш класс Blockchain:

```
1  import json
2  import random
3
4  from datetime import datetime
5  from hashlib import sha256
6
7
8  class Blockchain(object):
9      def __init__ (self):
10         self.chain = []
11         self.pending_transactions = []
12
13         # Создаем блок генезиса
14         print("Creating genesis block")
15         self.chain.append(self.new_block())
16
17     def new_block(self):
18         block = {
19             'index': len(self.chain),
20             'timestamp': datetime.utcnow().isoformat(),
21             'transactions': self.pending_transactions,
22             'previous_hash': self.last_block["hash"] if
```

```
                self.last_block else None,
23          'nonce': format(random.getrandbits(64), "x"),
24      }
25
26      # Получаем хэш этого нового блока и добавляем его в
          блок
27      block_hash = self.hash(block)
28      block["hash"] = block_hash
29
30      # Сброс списка незавершенных транзакций
31      self.pending_transactions = []
32
33      return block
34
35  @staticmethod
36  def hash(block):
37      # Мы гарантируем, что словарь отсортирован, иначе у нас
          будут несогласованные хэши
38      block_string = json.dumps(block, sort_keys=True).
        encode()
39      return sha256(block_string).hexdigest()
40
41  @property
42  def last_block(self):
43      # Возвращает последний блок в цепочке (если есть блоки)
44      return self.chain[-1] if self.chain else None
45
46  @staticmethod
47  def valid_block(block):
48      # Проверяет, начинается ли хеш блока с 0000
49      return block["hash"].startswith("0000")
50
```

```
51      def proof_of_work(self):
52          while True:
53              new_block  =  self.new_block()
54              if self.valid_block(new_block):
55                  break
56
57          self.chain.append(new_block)
58          print("Found a new block: ", new_block)
```

Нам нужно сделать два важных удаления:

- Удалите оператор печати внутри метода new_block(), иначе при майнинге вы получите много на консоли много нежелательных данных.
- В строке 33 внутри метода new_block() удалите строку self.chain.append(block), которая добавляет новый блок в цепочку (нам не нужны непроверенные блоки).

Теперь пришло время запустить ipython, создать экземпляр класса Blockchain и добыть несколько блоков:

```
In [1]: from blockchain import Blockchain

In [2]: bc = Blockchain()
Creating genesis block

In [3]: bc.proof_of_work()
Found a new block:
{
  "index": 1,
  "timestamp": "2020-02-07T04:39:03.920459",
  "transactions": [],
```

```
  "previous_hash": "1a93bd623f3e3aba2c9d4ee9193d36904382e7416b
  2fb854dbf3fc6a13cab702",
  "nonce":  "8b9991ef3ea9eb19",
  "hash": "0000075c4a6d3ae6ab06677887618cf41255d7a1ba10220bb
          3e9bc8c8ecec80e"
}

In [4]: bc.proof_of_work()
Found a new block:
{
  "index": 2,
  "timestamp": "2020-02-07T04:39:14.440383",
  "transactions": [],
  "previous_hash": "0000075c4a6d3ae6ab06677887618cf41255d7a1ba
                   10220bb3e9bc8c8ecec80e",
  "nonce": "b9a579d2cfa587b0",
  "hash": "00009d974a58fb689ad280c121897eb2f6da699db2c4bd2a3312
          dc41d478c182"
}

In [5]: bc.proof_of_work()
Found a new block:
{
  "index": 3,
  "timestamp": "2020-02-07T04:39:20.744179",
  "transactions": [],
  "previous_hash": "00009d974a58fb689ad280c121897eb2f6da699db2c
                   4bd2a3312dc41d478c182",
  "nonce":  "f4de39bb07bce043",
  "hash": "000003910a7ad733f89a9ff13ffea5b5e08fe69d581a4ecd30f
          237f95800221e"
}
```

Чтобы настроить сложность нашего майнинга, мы могли бы изменить количество ведущих нулей. Но и четырех достаточно. Вы обнаружите, что добавление одного начального нуля экспоненциально увеличивает время, необходимое для поиска решения. В биткойне сложность корректируется каждые 2016 блоков на основе консенсуса в сети. Сложность повышается, чтобы компенсировать неизбежный рост производительности аппаратного обеспечения.

Денежная масса

Теперь у вас есть полностью функционирующий класс блокчейна, реализующий майнинг для создания новых блоков. Это очень близко к тому, как происходит майнинг в производственных блокчейнах, таких как Биткойн. **Это алгоритм майнинга, который генерирует новый биткойн**: майнер, завершая добычу блока, вставляет транзакцию *ни от кого* самому себе с количеством биткойнов, которое уменьшается вдвое каждые 210 000 блоков. **Так появляются новые биткойны**.

ГЛАВА 5

Сеть

Это практическая, техническая глава о программировании сокетов. Мы узнаем, как отправлять и получать пакеты информации по сети Интернет путем создания удобного сетевого приложения в форме онлайн-чата, которое поможет нам получить некоторый опыт, прежде чем мы погрузимся в одноранговые сети.

Но так как эта книга не о написании хорошего кода на Python, мы отбросим осторожность и изучим теорию методом проб, ошибок и примеров. На протяжении всей этой главы я прошу вас терпеть меня и сосредоточиться на уровне 3 000 метров, а не на мелких деталях, поскольку это очень сложный материал. И держите под рукой свои навыки гугления, чтобы утолить жажду дополнительных разъяснений.

Краткое выражение благодарности за Интернет

Меня регулярно унижает Интернет, особенно когда я использую его на низком уровне — тот факт, что я могу открыть соединение в Нью-Йорке и отправить пакеты моему брату в Южную Африку с задержкой всего 180 мс, просто вдохновляет. — удивительно, что это ***просто*** работает.

Отказоустойчивость Интернета проистекает из простоты его основных протоколов (о чем мы и собираемся узнать больше), TCP (протокол управления передачей) и UDP (протокол пользовательских датаграмм). Вблизи они образуют простые

элементарные правила, но когда все участники соблюдают их на высоком уровне, протоколы создают надежную глобальную сеть, которую мы используем без особых раздумий, когда отправляем электронные письма, общаемся посредством видеоконференций или отправляем биткойны друг другу.

Важно понимать, что сетевые протоколы аналогичны луковице: Интернет состоит из уровней абстракции — электрические сигналы объединяются в группы; а группы образуют кадры, которые, в свою очередь, объединяются в пакеты данных; и так далее. Эти кадры определяются протоколами, которые определяют, как они выглядят и как используются. Сами протоколы определяются многолетними исследованиями, кульминацией которых являются документы с открытым исходным кодом, называемые RFC (запрос на комментарии), и имеют богатую историю. Я призываю вас прочитать их. Для начала вот RFC UDP (протокол пользовательских датаграмм), определенный в 1980 году: https://tools.ietf.org/html/rfc768.

Вот как примерно выглядит сетевой стек с точки зрения уровней:

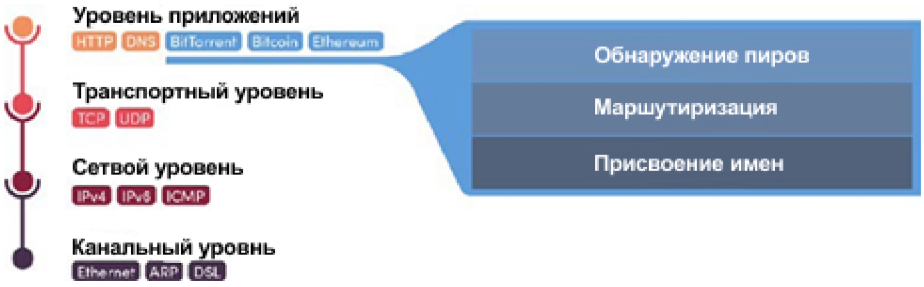


Рис. 5-1. *Уровни сетевого стека*

- Протоколы **уровня приложений** ближе всего к пользователю и логически определяют, как пользователи/клиенты взаимодействуют со службой, такой как веб-сайт (HTTP).
- **Транспортный уровень** инкапсулирует прикладной уровень — сообщения, отправляемые на прикладном

уровне, разбиваются на сегменты и пакеты и устанавливают соединение (транспорт) между двумя хостами в сети.

- **Сетевой уровень** инкапсулирует транспортный уровень и абстрагирует понятие «сети» — Интернет состоит из множества взаимосвязанных сетей, и этим сетям требуется протокол для маршрутизации данных между ними.
- **Канальный уровень** отвечает за передачу необработанных неструктурированных данных посредмтвом физической среды, такой как оптоволоконный кабель. В примере с оптоволоконным кабелем он отвечает за преобразование цифровых битов в импульсы света.

В этой главе мы сосредоточимся на транспортном уровне, который позволяет нам отправлять и получать информацию между двумя хостами в сети Интернет. Это означает, что мы будем использовать два основных протокола, используемых на транспортном уровне: UDP и TCP. Основное различие между ними заключается в том, что TCP **поддерживает состояние**, а это означает, что соединение устанавливается между двумя хостами до того, как информация пойдет в любом направлении — как и при телефонном звонке, оба хоста остаются подключенными друг к другу. Соединение остается открытым до тех пор, пока не будет отправлен сигнал закрытия.

UDP, с другой стороны, является протоколом **без сохранения состояния**, что означает, что пакет информации *отправляется* получателю, и отправитель не ждет подтверждения; вот почему UDP часто называют протоколом «выстрелил-забыл» — это похоже на отправку почты кому-то, пользующемуся почтовой службой, но гораздо быстрее.

Параллельное программирование на Python

Сетевое программирование относится к области *параллельного* программирования, что является сложной концепцией для начинающего программиста. Это связано с тем, что программам, требующим работы в сети, обычно приходится выполнять множество операций одновременно; например, чат-сервер должен взаимодействовать с несколькими клиентами одновременно, или биткойн-узел должен взаимодействовать с несколькими одноранговыми узлами одновременно.

Существует *множество* способов борьбы с параллелизмом, два самых популярных из них — это **потоки** (несколько путей выполнения внутри запущенного процесса), либо полностью **параллельные** процессы. Но мы собираемся использовать другой подход, называемый *совместной многозадачностью*, широко известный как «асинхронный». В Python асинхронное программирование облегчается библиотекой **asyncio**.

Асинхронное программирование достигается за счет достаточно быстрой приостановки и возобновления выполнения, чтобы программа выглядела так, *как будто* она выполняется параллельно. Но на самом деле это просто быстрое переключение между различными командами с вычислением крошечного фрагмента каждой программы перед переключением на другую. Это похоже на нарезку овощей, за исключением того, что вместо того, чтобы полностью нарезать лук, а затем перейти к картофелю, вы нарезаете немного лука, а затем немного картофеля и наоборот, пока оба овоща не будут нарезаны.

Вы можете подумать, что это влияет на производительность, так как наша программа на самом деле не работает параллельно, но для сети (или операций ввода-вывода) она оказывается чрезвычайно производительной и позволит нашему серверу обрабатывать гигантское количество подключений одновременно.

Асинхронный код против многопоточности

Создание асинхронного кода — чрезвычайно популярный подход для современных веб-приложений. Основная причина этого заключается в том, что вы можете писать производительный код без сохранения состояния с минимальной вероятностью возникновения условий конкуренции. Node.js, например, пишется асинхронно в одном потоке. Многопоточный код, с другой стороны, трудно тестировать и он требует тщательного планирования архитектуры, с целью недопущения возникновения условий конкуренции.

Быстрое введение в asyncio

Библиотека Python **asyncio** позволяет нам писать асинхронный код в удобном для чтения виде, предоставляя нам массу вспомогательных функций. Это позволяет нам разделить наш код на задачи, которые выглядят так, как будто они выполняются параллельно.

Давайте разберемся, как работает **asyncio**, взглянув на очень простую программу:

```
1  import asyncio
2  import time
3
4
5  async def greet(name, delay):
6      await asyncio.sleep(delay)
7      print(f'{name}: I waited {delay} seconds before saying
       "hello"')
8
9
10 async def main():
11     task_1 = syncio.create_task(greet("t1", 3))
```

```
12      task_2 = syncio.create_task(greet("t2", 2))
13      task_3 = syncio.create_task(greet("t3", 2))
14
15      start_time = time.time()
16
17      print("0.00s: Program  Start")
18
19      await task_1
20      await task_2
21      await task_3
22
23  print(f"{time.time() - start_time:.2f}s: Program End")
24
25
26      asyncio.run(main())
```

Наиболее важными фрагментами этой программы являются строки:

```
1  task_1 = asyncio.create_task(greet(1))
2  task_2 = asyncio.create_task(greet(2))
3  task_3 = asyncio.create_task(greet(2))
```

которые «превращают» три задачи — функции с аргументами — в задачи, которые могут выполняться одновременно; они называются *ожидаемыми*, так как их скоро будут «ождать»

Следующие строки

```
1  await task_1
2  await task_2
3  await task_3
```

отвечают за запуск задач, которые должны быть запущены с использованием выражения ожидания Python.

Наконец, `main()` вызывается внутри функции `run()` **asyncio**; это говорит Python, что `main` (и операторы внутри него) должны выполняться одновременно.

При запуске программа выводит следующее:

```
0.00s: Program Start
t2: I waited 2 seconds before saying "hello"
t3: I waited 2 seconds before saying "hello"
t1: I waited 3 seconds before saying "hello"
3.00s: Program End
```

Как показано, наши задачи действительно выполнялись одновременно; если бы программа работала ***синхронно***, то ее выполнение заняло бы около 7 секунд, так как каждая функция выполнялась бы последовательно.

Во введении я упомянул, что мы «приостанавливаем» наши задачи; здесь это делается строкой `await asyncio.sleep(...)` внутри нашей функции. Найдите минутку, чтобы изучить это; она приводит к следующему вопросу:

"Если задержка не достигнута, приостановите функцию (ожидание) здесь и возобновите ее чем-либо другим".

Крайне важно понять этот шаблон концептуально (даже если вы не полностью его понимаете) — и опробовать его самостоятельно — потому что он формирует шаблон большинства асинхронных программ. Главную функцию иногда называют ***точкой входа*** в нашу программу, и мы будем широко использовать ее во время разработки.

Полный курс по **asyncio** выходит далеко за рамки этой книги, а в контексте построения блокчейнов это одновременно важно и неважно. Важно понимать, что происходит внутри (если вы заботитесь о создании готовых коммерческих приложений), но это также неважно в контексте изучения блокчейнов и сетей, потому что на самом деле это детали реализации, о которых мы сможем узнать больше позже.

Создание чат-сервера с нуля

Поскольку построение блокчейна довольно бесполезно, если он не подключен к другим клиентам в сети, мы собираемся начать с чего-то похожего и немного с чат-сервера, который позволяет множеству подключенных клиентов общаться друг с другом одновременно. Вы даже можете использовать его для размещения собственного чат-сервера и подключения к нему друзей со всего мира. Мы сделаем это примерно в 100 строках на Python.

Когда я рос в конце 90-х, я использовал IRC (Internet Relay Chat). Это надежный протокол чата раннего Интернета, который позволял людям со всего мира общаться в чатах. На самом деле, канал #bitcoin в сети freenode до сих пор активно используется — именно здесь общаются все основные участники. Но внедрение протокола IRC — довольно сложная задача, и, поскольку это «обучение на практике», мы проигнорируем всю предыдущую работу и во имя экономии времени и простоты сосредоточимся на компактном примере, который поможет понять суть создания асинхронных клиент-серверных TCP-приложений на Python.

Современные веб-приложения являются масштабируемыми и параллельными, поэтому важно понимать, как пишется асинхронный код, чтобы избежать распространенных ошибок.

Для достижения нашей цели мы собираемся создать наш чат-сервер с использованием сокетов TCP. Python упрощает работу с ними. На самом деле, модуль **asyncio** предоставляет множество функций по умолчанию.

Мы начнем с самого простого — простого «эхо-сервера», который просто отправляет обратно любое отправленное ему сообщение.

```
1 import asyncio
2
3
4 async def handle_connection(reader, writer):
```

```
5       writer.write("Hello new user, type something...\n".
        encode())
6
7       data = await reader.readuntil(b"\n")
8
9       writer.write("You sent: ".encode() + data)
10      await writer.drain()
11
12      # Давайте закроем соединение и сбросим его
13      writer.close()
14      await writer.wait_closed()
15
16
17  async def main():
18      server = await asyncio.start_server(handle_
        connection,  "0.0.0.0", 8888)
19
20      async with server:
21          await server.serve_forever()
22
23
24      asyncio.run(main())
```

Прежде чем мы разберем код, давайте запустим этот сервер и отправим ему некоторую информацию, чтобы получить представление об ожиданиях. Откройте два окна терминала и расположите их рядом друг с другом.

В первом окне терминала запустите предыдущий код:

```
$ python my_server.py
```

А во втором окне терминала подключитесь к серверу с помощью nc

```
nc 127.0.0.1 8888
```

или telnet, если вы находитесь в среде Windowss:

```
telnet 127.0.0.1 8888
```

Примечание для разных ОС

И nc (netcat), и telnet — это программы, которые позволяют открывать сокет-соединения с удаленными хостами. Поскольку я не уверен, какую операционную систему вы используете, я оставлю этот вопрос вам, чтобы вы выяснили, какую из них использовать (или установить, если это необходимо).

Введите сообщение и нажмите Enter; вы должны увидеть следующее:

```
nc 127.0.0.1 8888

Hello new user, type something...
hey fellow blockchain enthusiast!
You sent: hey fellow blockchain enthusiast!
```

Если вы видите это сообщение, отлично — вы можете подключиться к нашему серверу по IP-адресу 127.0.0.1 (это специальный локальный IP-адрес, назначенный вашему собственному компьютеру) и порту 8888 (порт, который мы выбрали). Если вы находитесь в локальной сети, вы можете попробовать подключиться к своему серверу с другого компьютера; просто замените 127.0.01 на ваш локальный IP-адрес.

Застряли? Убедитесь, что это не так, прежде чем продолжить

Если вы не видете выше указанное сообщение, вам нужно исправить ситуацию, прежде чем продолжать. Убедитесь в том, что код вашего сервера Python действительно работает и что никакой другой процесс не использует порт 8888.

В предыдущем коде сервер инициализируется (строка 18) *функцией обратного вызова* handle_connection для обработки новых

подключений. Функция соединения `handle_connection` неявно получает данные чтения и записи в качестве аргументов (строка 4), представляющих базовое соединение. В строке 24 мы инструктируем сервер запуститься и никогда не останавливаться.

Поддержка нескольких подключений

Поскольку мы используем asyncio, попробуйте открыть несколько терминальных окон и подключиться к своему серверу. Вы обнаружите, что, поскольку наш код является асинхронным, мы можем поддерживать множество одновременных подключений. Сколько? Максимальное количество клиентов теоретически — это максимальное количество портов, которое может назначить ваша операционная система; в большинстве систем это примерно 65 536 (или 2^{16}), но вы, вероятно, будете ограничены памятью и непроизводительными затратами процессора задолго до того, как достигнете этого количества.

Создание чат-сервера

Мы создадим чат-сервер, установив сначала для чата на нашем сервере простой *протокол* связи:

- Когда пользователь подключается, у него должен быть запрошен его *псевдоним*.
- Когда пользователь подключается, его появление должно транслироваться всем подключенным пользователям (кроме него самого).
- Если пользователь отправляет какое-либо сообщение, его сообщение транслируется всем подключенным пользователям (кроме него самого).
- Если пользователь отправляет сообщение `/list`, он должен увидеть список всех подключенных пользователей.

- Если пользователь отправляет сообщение /quit, он должен быть отключен, и всем подключенным пользователям должно быть передано сообщение "<nickname> hasquit".

Давайте создадим класс ConnectionPool для управления «пулом» подключенных клиентов и разместим логику для упомянутого протокола. Для этого мы создадим несколько шаблонов подстановки для методов (я добавил несколько аннотаций, объясняющих, что должен делать каждый метод).

```
1  import asyncio
2
3
4  class ConnectionPool:
5      def __init__(self):
6          self.connection_pool = set()
7
8      def send_welcome_message(self, writer):
9          """
10         Sends a welcome message to a newly connected client
11         """
12         pass
13
14     def broadcast(self, writer, message):
15         """
16         Broadcasts a general message to the entire pool
17         """
18         pass
19
20 def broadcast_user_join(self, writer):
21     """
22     Calls the broadcast method with a "user joining"
       message
23     """
```

```
24      pass
25
26  def broadcast_user_quit(self, writer):
27      """
28      Calls the broadcast method with a "user quitting"
        message
29      """
30      pass
31
32      def broadcast_new_message(self, writer, message):
33          """
34          Calls the broadcast method with a user's chat message
35          """
36          pass
37
38      def list_users(self,writer):
39          """
40          Lists all the users in the pool
41          """
42          pass
43
44      def add_new_user_to_pool(self,writer):
45          """
46          Adds a new user to our existing pool
47          """
48          self.connection_pool.add(writer)
49
50      def remove_user_from_pool(self, writer):
51          """
52          Removes an existing user from our pool
53          """
54          self.connection_pool.remove(writer)
55
56
```

```
57 async def handle_connection(reader, writer):
58     writer.write("Hello new user, type something...\n".
       encode())
59
60     data = await reader.readuntil(b"\n")
61
62     writer.write("You sent: ".encode() + data)
63     await writer.drain()
64
65     # Давайте закроем соединение и сбросим его
66     writer.close()
67     await writer.wait_closed()
68
69
70 async def main():
71     server = await asyncio.start_server(handle_connection,
       "0.0.0.0", 8888)
72
73     async with server:
74         await server.serve_forever()
75
76
77 connection_pool = ConnectionPool()
78 asyncio.run(main())
```

Тех, кто не знаком с asyncio, может смутить архитектура нашего класса ConnectionPool. Он создается только один раз, и его методы принимают аргумент с именем writer. Этот аргумент writer является экземпляром StreamWriter —объекта asyncio, который отвечает за запись в базовое соединение: подключенного пользователя. Он обрабатывается «одновременно», потому что для каждого нового соединения нашему handle_connection создается новый экземпляр writer. Думайте о writer как о подключенном пользователе.

Найдите минутку, чтобы изучить эти шаблоны методов и подумать, как мы можем начать их заполнять — это проще, чем может показаться.

Мы начнем с двух вещей:

1. Сбор псевдонимов пользователей
2. Заполнение метода send_welcome_message()

```
1       import asyncio
2       from textwrap
3       import dedent
4
4
5       class ConnectionPool:
6   def __init__(self):
7       self.connection_pool = set()
8
9   def send_welcome_message(self, writer):
10      message = dedent(f"""
11      ===
12      ( Welcome {writer.nickname}!
13
14      There are {len(self.connection_pool) - 1} user(s)
        here beside you
15      ===
16      """)
17
18              writer.write(f"{message}\n".encode())
19
20        def broadcast(self, writer, message):
21              pass
22
23        def broadcast_user_join(self, writer):
24              pass
```

```
25
26      def broadcast_user_quit(self, writer):
27          pass
28
29      def broadcast_new_message(self, writer, message):
30          pass
31
32      def list_users(self, writer):
33          pass
34
35      def add_new_user_to_pool(self, writer):
36          self.connection_pool.add(writer)
37
38      def remove_user_from_pool(self, writer):
39          self.connection_pool.remove(writer)
40
41
42 async def handle_connection(reader, writer):
43     # Получаем псевдоним для нового клиента
44     writer.write("> Choose your nickname: ".encode())
45
46     response = await reader.readuntil(b"\n")
47     writer.nickname = response.decode().strip()
48
49 connection_pool.add_new_user_to_pool(writer)
50 connection_pool.send_welcome_message(writer)
51 await writer.drain()
52
53 # Давайте закроем соединение и сбросим его
54 writer.close()
55 await writer.wait_closed()
56
57
58     async def main():
```

```
59 server = await asyncio.start_server(handle_connection,
   "0.0.0.0", 8888)
60
61 async with server:
62     await server.serve_forever()
63
64
65     connection_pool = ConnectionPool()
66     asyncio.run(main())
```

Мы собираем псевдонимы пользователей в строках 67—71 и сохраняем их как атрибут объекта записи в строке 72. Обратите особое внимание на то, как мы «читаем до тех пор, пока» новый символ строки не будет введен пользователем в строке 71.

Далее мы транслируем приветственное сообщение в строках 10—19.

На этом этапе вы должны запустить сервер и подключиться к нему, чтобы получить приветственное сообщение. На самом деле, удобно запускать сервер после внесения каждого изменения, чтобы упростить отладку.

Следующий код заполняет остальные методы и завершает работу сервера:

```
1  import asyncio
2  from textwrap import dedent
3
4
5  class ConnectionPool:
6      def __init__(self):
7          self.connection_pool = set()
8
9      def send_welcome_message(self, writer):
10         message = dedent(f"""
11         ===
12         Welcome {writer.nickname}!
```

```
13
14          There are {len(self.connection_pool) - 1} user(s)
            here beside you
15
16          Help:
17           - Type anything to chat
18           - /list will list all the connected users
19           - /quit will disconnect you
20          ===
21          """)
22
23          writer.write(f"{message}\n".encode())
24
25      def broadcast(self, writer, message):
26          for user in self.connection_pool:
27              if user != writer:
28                  # Нам не нужно также транслировать на
                    пользователя, отправляющго сообщение
29  user.write(f"{message}\n".encode())30
31      def broadcast_user_join(self, writer):
32          self.broadcast(writer, f"{writer.nickname} just
              joined")
33
34      def broadcast_user_quit(self, writer):
35          self.broadcast(writer, f"{writer.nickname}
            just quit")
36
37      def broadcast_new_message(self, writer, message):
38          self.broadcast(writer, f"[{writer.nickname}]
            {message}")
39
40      def list_users(self,  writer):
41          message = "===\n"
42          message += "Currently connected users:"
```

```
43          for user in self.connection_pool:
44              if user == writer:
45                  message += f"\n - {user.nickname} (you)"
46              else:
47                  message += f"\n - {user.nickname}"
48
49          message += "\n==="
50          writer.write(f"{message}\n".encode())
51
52      def add_new_user_to_pool(self, writer):
53          self.connection_pool.add(writer)
54
55      def remove_user_from_pool(self, writer):
56          self.connection_pool.remove(writer)
57
58
59  async def handle_connection(reader, writer):
60      # Получние псевдонима для нового клиента
61      writer.write("> Choose your nickname: ".encode())
62
63      esponse = await reader.readuntil(b"\n")
64      writer.nickname = response.decode().strip()65
66      connection_pool.add_new_user_to_pool(writer)
67      connection_pool.send_welcome_message(writer)
68      await writer.drain()69
70      # Давайте закроем соединение и сбросим его
71      writer.close()
72      await writer.wait_closed()73
74
75      async def main():
76      server = await      asyncio.start_server(handle_connection,
        "0.0.0.0", 8888)
77
```

```
78      async with server:
79      await server.serve_forever()80
81
82      connection_pool = ConnectionPool()
83      asyncio.run(main())
```

Давайте попробуем подключиться к нашему серверу и убедиться, что он работает должным образом. Во-первых, убедитесь, что вы используете сервер с последними дополнениями к коду. Теперь откройте окно терминала и подключитесь с помощьюnc (или `telnet`):

```
nc 127.0.0.1 8888

> Choose your nickname: blockchain_dan

===

Welcome blockchain_dan!

There are 0 user(s) here beside

youHelp:

- Type anything to chat
- /list will list all the connected users
- /quit will disconnect you
===
```

Как и ожидалось, после подключения сервер запрашивает у нас наш псевдоним (“blockchain_dan”) и показывает нам приветственное сообщение, а затем отключает нас. Пришло время ввести цикл с некоторой логикой, чтобы держать пользователя на связи:

```
1  import asyncio
2  from textwrap import dedent
3
4
```

```
class ConnectionPool:
    def __init__ (self):
        self.connection_pool = set()

    def send_welcome_message(self, writer):
        message = dedent(f"""
        ===
        ( Welcome {writer.nickname}!

        There are {len(self.connection_pool) - 1} user(s)
        here beside you

        Help:
         - Type anything to chat
         - /list will list all the connected users

         - /quit will disconnect you
        ===
        """)

        writer.write(f"{message}\n".encode())

    def broadcast(self, writer, message):
        for user in self.connection_pool:
            if user != writer:
                # We don't need to also broadcast to the
                    user sending the message
user.write(f"{message}\n".encode())
    def broadcast_user_join(self, writer):
        self.broadcast(writer, f"{writer.nickname just
          joined")

    def broadcast_user_quit(self, writer):
        self.broadcast(writer, f"{writer.nickname} just
```

```
            quit")
36
37      def broadcast_new_message(self, writer, message):
38          self.broadcast(writer, f"[{writer.nickname}]
              {message}")
39
40      def list_users(self,     writer):
41          message = "===\n"
42          message += "Currently connected users:"
43          for user in self.connection_pool:
44              if user == writer:
45                  message += f"\n - {user.nickname} (you)"
46              else:
47                  message += f"\n - {user.nickname}"
48
49          message += "\n==="
50          writer.write(f"{message}\n".encode())
51
52      def add_new_user_to_pool(self, writer):
53          self.connection_pool.add(writer)
54
55      def remove_user_from_pool(self, writer):
56          self.connection_pool.remove(writer)
57
58
59  async def handle_connection(reader, writer):
60      # Получние псевдонима для нового клиента
61      writer.write("> Choose your nickname: ".encode())
62
63      response = await reader.readuntil(b"\n")
64      writer.nickname = response.decode().strip()
65
66      connection_pool.add_new_user_to_pool(writer)
67      connection_pool.send_welcome_message(writer)
```

```
68
69      # Объявление о прибытии этого нового пользователя
70      connection_pool.broadcast_user_join(writer)
71
72      while True:
73          try:
74              data = await reader.readuntil(b"\n")
75          except asyncio.exceptions.IncompleteReadError:
76              connection_pool.broadcast_user_quit(writer)
77              break
78
79          message = data.decode().strip()
80              if message == "/quit":
81                  connection_pool.broadcast_user_quit(writer)
82                  break
83              elif message == "/list":
84                  connection_pool.list_users(writer)
85              else:
86                  connection_pool.broadcast_new_message
                    (writer, message)
87
88 await writer.drain()89
90              if writer.is_closing():
91                  break
92
93          # Теперь мы вышли из цикла сообщений, и пользователь
              вышел. Давайте сбросим...
94          writer.close()
95          await writer.wait_closed()
96 connection_pool.remove_user_from_pool(writer)97
98
99 async def main():
100         server = await asyncio.start_server(handle_
            connection, "0.0.0.0", 8888)
```

```
101
102         async with server:
103             await server.serve_forever()
104
105
106     connection_pool = ConnectionPool()
107     asyncio.run(main())
```

Организуем петлю на строках 72—91. Обратите внимание, как мы размещаем пользовательский ввод (строка 74) в блок `try-except`, чтобы обслуживать пользователя, чье соединение обрывается без предупреждения. Затем мы продолжаем обрабатывать их сообщение — проверяя, является ли это методом `quit` или `list` — пока снова не дождемся дальнейшего ввода данных.

На данный момент наш чат-сервер полностью функционален. Давайте откроем два или три окна терминала, чтобы имитировать настоящий чат:

Рис. 5-2. *Два терминала, расположенные рядом, имитирующие общение двух пользователей друг с другом*

Примечание попросите друга подключиться к вашему серверу в локальной сети, предоставив ему IP-адрес вашего компьютера. Вы также можете разместить свой чат-сервер на веб-хостинге и привязать к нему полный URL-адрес, чтобы любой пользователь в Интернете мог подключиться к нему.

Важно экспериментировать и практиковаться с концепциями, продемонстрированными в этом пошаговом руководстве, поскольку они будут играть решающую роль в достижении цели этой книги: основополагающий сетевой уровень для нашей сети блокчейнов.

Совет: полный исходный код находится в репозитории github по адресу https://github.com/dvf/blockchain-book.

Протоколы

Протоколы чрезвычайно важны при проектировании одноранговых сетей. Протоколы — это «правила игры», и их сложно разработать, поскольку они требуют долгосрочного предвидения и планирования. Отсутствие надежной архитектуры приводит к архитектурным разногласиям в будущем — это проблемам, которые требуют так называемых «хардфорков»; или изменения базового протокола сети. Хорошее планирование оставляет достаточно места для будущих обновлений, будь то технологические достижения или социальные изменения в сети.

Частью успеха Биткойн является простота его протокола и тщательный подход его основных разработчиков при внесении

новых изменений. Разногласия на уровне протокола часто приводили к «хардфоркам» по инициативе сообщества, таким как Bitcoin Cash и множеству ответвлений блокчейна Биткойн.

Интересно изучить, почему и когда произошли эти ответвления, потому что они почти всегда являются результатом разногласий на уровне протокола.

Имея это в виду, давайте посмотрим на созданный нами чат-сервер и проанализируем его протокол, взглянув на возможные «сообщения». Полезно разбить их на «пользовательские истории»:

- Как подключенный пользователь, я могу выйти, отправив сообщение `/quit`.
- Как подключенный пользователь, я могу перечислить всех подключенных пользователей, отправив сообщение `/list`.
- Как для подключенного пользователя, любой текст (за исключением упомянутых выше сообщений), который я отправляю, транслируется всем подключенным клиентам.

Наш игрушечный чат-сервер необычайно прост. Но, определив предыдущий протокол, мы можем сделать его ***универсальным***; это очень важно, потому что это означает, что ***любой*** клиент в сети Интернет, использующий свое собственное программное обеспечение, должен всего лишь следовать протоколу, чтобы успешно взаимодействовать с нашим сервером.

Внедряя протокол Биткойн, разработчики могут писать собственное программное обеспечение для взаимодействия с сетью. Им нужно только изучить, какие сценарии и сообщения возможны в сети. Wiki сети Биткойн достоверно отображает эту информацию по этому адресу: https://en.bitcoin.it/wiki/Protocol_documentation. Когда вы загружаете программное обеспечение под названием «Bitcoin Core», вы на самом деле загружаете ***эталонную*** реализацию протокола, разработанную и поддерживаемую разработчиками, которые посвящают все свое

свободное время Биткойну.

Основа для создания блокчейна

Одноранговые сети становятся возможными благодаря тому, что каждый клиент в сети коллективно реализует выбранный протокол. В качестве упражнения остановитесь на мгновение и попытайтесь выяснить, какие сообщения должны быть допустимыми в одноранговой сети.

Следуя совету, полезно сначала разбить нашу сеть на пользовательские сценарии, чтобы прояснить, какие сообщения необходимы. Давайте определим понятие *подключенного* узла как узела, обладающего достаточным числом связей с достаточным количеством других одноранговых узлов для формирования одноранговой сети. Вот несколько возможных сценариев:

- Как *узел*, я могу подключаться к пирам, обнаруживая их*.
- Будучи *подключенным узлом*, я могу опубликовать список своих одноранговых узлов для всех, кто их запрашивает.
- Как *подключенный узел*, я могу принимать и транслировать новую транзакцию от однорангового узла.
- Будучи *подключенным узлом*, я могу передать содержимое блока любому запрашивающему узлу.
- Как *подключенный узел* я могу принять новый блок и добавить его в свою цепочку блоков, если он соответствует определенным критериям.

Gossip

Предыдущий список никоим образом не является исчерпывающим — мы не учитывали штрафные меры для одноранговых узлов, которые плохо себя ведут или формируют какое-либо разрешающее решение в том случае, когда необходимо добавить

два допустимых (конфликтующих) блока, — но на данный момент это хорошая основа. чтобы строить. Но самое главное, так как нет централизованного уравления, нам нужен способ, чтобы пир локализовал сеть и сформировал *рой*.

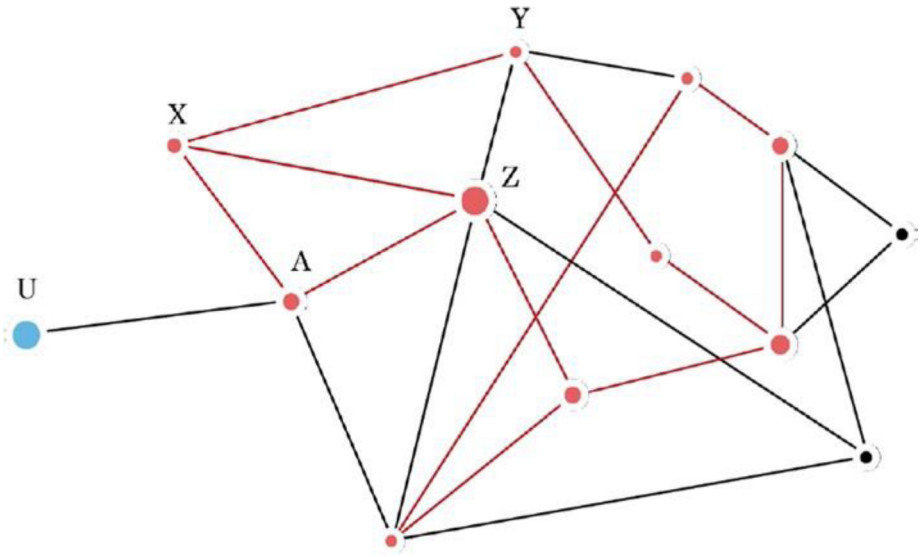


Рис. 5-3. *Узел, впервые подключающийся к сети*

На предыдущем изображении узел U должен получить список других узлов (равноправных узлов) от узла A, а узел A, в свою очередь, получил список узлов от своих соседей X, Y и Z и должен постоянно «опрашивать» их чтобы убедиться, что они все еще «живы» в сети. А также он должен объявить о присутствии U в сети и так далее и тому подобное. Эта общая схема называется протоколом Gossip, и успешный протокол Gossip — это то, что формирт устойчивую децентрализованную сеть.

ГЛАВА 6

Криптография

Изучение алгоритмов, представленных в этой главе, выходит далеко за рамки любой книги, которую вы можете найти, или курса, который вы можете пройти. Они требуют многих лет изучения и практики и основаны на алгебраических понятиях, связанных с конкретными областями математики. Но, к счастью, теоретическое понимание того, как работают криптографические алгоритмы, не является обязательным требованием для использования этих инструментов на практике, если вы понимаете их назначение и результаты. Другими словами, здесь важны практические знания о том, как использовать криптографические библиотеки.

Если вы похожи на меня (я не криптограф), правильный подход к этому материалу — это осторожность. Я достаточно разбираюсь в криптографии, чтобы понять, что это постоянный баланс — исследование компромиссов между безопасностью, удобством и скоростью — с изменением векторов атак по мере совершенствования технологий. Как человек, изучающий этот материал впервые, вы должны склоняться в пользу проверенных в бою, хорошо известных лучших алгоритмов и инструментов. В нашем цифровом мире слишком многое поставлено на карту, чтобы идти коротким путем — большинство взломов и атак в Интернете вызваны простыми ошибками: использованием плохих реализаций не совсем понятных вещей или, что еще хуже, личных пристрастий. В мире криптографии есть поговорка: *никогда не создавайте собственный бренд криптографии*.

Криптография — это *крипто* в криптовалюте. И настоящих криптографов может раздражать использование слова «крипто» для обозначения криптовалюты. Подписание чека в реальной жизни очень похоже на подписание транзакции в блокчейне, но еще более безопасно, поскольку подпись нельзя подделать. На самом деле, она меняется в зависимости от документа, который вы подписываете!

Отправка сообщений с целостностью

Прежде чем мы углубимся в цифровые подписи и криптографию с открытым ключом, я хочу воспользоваться вашими знаниями о хешировании, показав вам, как их можно использовать для отправки не поддающихся подделке сообщений через небезопасную среду, такую как Интернет.

Примечание Обратите внимание, что этот пример — всего лишь учебный пример, а для этого есть хорошо построенные библиотеки. Фактически, библиотека hmac, поставляемая с Python, создана именно для этой цели.

Допустим, Алиса хочет отправить сообщение Бобу. А также допустим, что Алиса и Боб перед отправкой сообщений договорились поделиться друг с другом секретным паролем p@55w0rd.

Алиса создает сообщение и объединяет его с p@55w0rd; затем она вычисляет хэш сообщения:

```
from hashlib import sha256

message = "Hello Bob, Let's meet at the Kruger National Park on
2020-12-12 at 1pm."
hash_message = sha256(("p@55w0rd" + message).encode()).
hexdigest()
```

Вычисленный хеш 39aae6ffdb3c0ac1c1cc0f50bf08871a729052

cf1133c4c9b44a5bab8fb66211. Затем Алиса отправляет это сообщение, включая хеш, Бобу, который теперь проверяет тот факт, что это сообщение могла отправить только Алиса:

```
from hashlib import sha256

alices_message = "Hello Bob, Let's meet at the Kruger
NationalPark on 2020-12-12 at 1pm."
alices_hash = "39aae6ffdb3c0ac1c1cc0f50bf08871a729052cf1133c4c
9b44a5bab8fb66211"
hash_message = sha256(("p@55w0rd" + alices_message).encode()).
hexdigest()

if hash_message == alices_hash:
    print("Message has not been tampered with")
```

Это тривиальный пример цифровой подписи, чтобы дать вам представление о том, как она работает. Вскоре мы увидим, что библиотеки, отвечающие за проверку и подпись сообщений, более надежны и не требуют использования предварительного выбранного общего ключа.

Симметричная криптография

Симметричная криптография является старейшим способом криптографии. Она включает в себя обмен ***общим*** ключом (шифром) с человеком, с которым вы хотите общаться.

Например, допустим, вы хотите поделиться своей машиной со своим братом. Вы делаете копию ключа и отдаете ее ему. Теперь только вы и ваш брат можете завести машину. Но пока вы в отпуске, вам звонит брат и сообщает, что ключ потерян или, возможно, его кто-то украл.

Этот пример иллюстрирует некоторые проблемы симметричной криптографии — основная из которых заключается в том, что вы доверяете второй стороне, а также в накладных

расходах и ведении списка уполномоченных лиц, которые могут управлять вашей машиной. Кроме того, вы можете не знать, когда конкретный ключ был скомпрометирован. Давайте рассмотрим классический пример симметричной криптографии.

Шифр Цезаря

В Риме около 50 г. до н.э. Юлий Цезарь использовал метод шифрования для своей личной переписки. Это простой метод симметричного шифрования, который работает путем сдвига символов в сообщении на фиксированную величину.

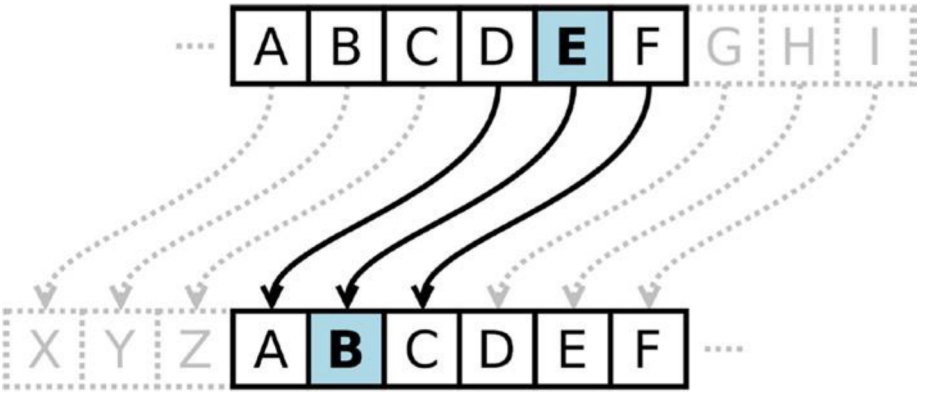


Рис. 6-1. *Шифр Цезаря*

Цезарь заранее давал получателю шифр (ключ), скажем, 3. В зашифрованном сообщении буква D становилась буквой A. Получатель применял обратны сдвиг, и буква A становилась снова буквой D.

Проблема здесь заключается в том, что акт отправки шифра небезопасен — любую форму связи можно проследить и установить ключ. Нам нужна форма шифрования, свободная от процесса создания и отправки общих ключей. Вот почему была изобретена криптография с открытым ключом, которая решает проблему общего доверия.

Криптография с открытым ключом

Криптография с открытым ключом является примером *асимметричной* криптографии. Это тип криптографии, который защищает почти все современные системы в Интернете. Мы рассмотрим его в упрощенном виде, чтобы вы поняли, как и почему это работает, а затем перейдем к техническим аспектам и деталям, разбросанным по нескольким примерам.

Криптография с открытым ключом включает не одну, а две (или более) пары ключей, одна из которых держится в секрете. Существует множество различных алгоритмов криптографии с открытым ключом, но мы будем использовать популярный алгоритм RSA (Rivest, Shamir, Adleman — создатели) (в случае Биткойна вместо него используется ECDSA (алгоритм цифровой подписи на эллиптических кривых)). Эти алгоритмы представляют собой сложные математические драгоценности, выходящие за рамки этой книги, они поставляются в различных вариантах, которые ценятся за определенные свойства: некоторые считаются устойчивыми к квантовым вычислениям; другие выбираются за их роизводитльность или легкость. Но результат любого из этих алгоритмов один и тот же: ***коррелированная*** пара ключей, A и B, которую вы можете использовать для шифрования сообщения, которое будет отправлено по незащищенному каналу.

Обратите внимание на слово коррелированный — ключи A и B связаны математическим способом. A должен храниться в секрете, и быть известным только вам. A B — это ваш открытый ключ: вы можете опубликовать его в Интернете, и кто-то может использовать его для шифрования сообщения, которое сможете расшифровать только вы (используя ключ A). Вы можете отправить свой ключ B любому, кто хочет общаться с вами безопасным способом, но вы должны хранить A, как Фродо хранил Кольцо. Мы постоянно используем криптографию с открытым

ключом — в основном, не осознавая этого — когда вы получаете доступ к веб-сайту через https, вы используете ключ B веб-сайта для шифрования исходящих данных, чтобы только веб-сайт мог их прочитать.

Кроме того — и это действительно важно — вы можете использовать свой закрытый ключ A для ***подписи*** сообщения, а ключ B (известный публике) может ***проверить*** подпись. Так проверяются транзакции в блокчейне. Но об этом чуть позже. Давайте сначала посмотрим на некоторые примеры, чтобы получить некоторое понимание.

Пример на Python

Вот фантастическая аналогия из отличной библиотеки Python NaCL с открытым исходным кодом:

> Представьте, что Алиса получает что-то ценное, отправленное ей. Она хочет убедиться, что ценность доставлена в целости и сохранности (т. е. не было вскрытия или подделки) и что она не является подделкой (т. е. что это на самом деле пришло от того отправителя, от которого она ожидает, и никто не подменил ее по дороге).
>
> Один из способов сделать это — дать отправителю (назовем его Боб) коробку с высокой степенью защиты по своему выбору. Алиса предоставляет Бобу такую коробку и еще кое-что: замок, но замок без ключа. Алиса держит ключ от замка при себе. Боб может положить предметы в коробку, а затем закрыть ее на замок. Но как только замок защелкнется, ящик никто открыть не сможет кром того, у кого есть ключ, хранящийся у Алисы.
>
> Но вот поворот: Боб также вешает замок на коробку. В этом навесном замке используется ключ, который Боб

опубликовал для всего мира, так что если у вас есть один из ключей Боба, вы знаете, что коробка пришла от него, потому что ключи Боба открывают замки Боба (давайте представим себе мир, в котором замки нельзя подделать, даже если вы знаете ключ). Затем Боб отправляет коробку Алисе.

Чтобы Алиса могла открыть ящик, ей нужны два ключа: ее секретный ключ, открывающий ее собственный замок, и всем известный ключ Боба. Если ключ Боба не открывает навесной замок Боба, Алиса знает, что это не та коробка, которую она ожидала от Боба, а подделка.

Эта двусторонняя гарантия идентификации известна как взаимная аутентификация.

Мы собираемся использовать PyNaCl, чтобы написать пример предыдущей аналогии на Python. Перво-наперво, давайте установим PyNaCl:

Примечание Кстати, PyNaCl — отличный пример хорошо протестированной библиотеки для использования в ваших собственных проектах. Самое главное — для криптографической библиотеки — у нее отличная документация, изобилующая примерами. Вы можете прочитать больше об этом здесь: https://pynacl.readthedocs.io/en/stable/public/

```
poetry add pynacl
```

Затем давайте активируем нашу виртуальную среду и создадим интерпретатор Python:

```
poetry shell
ipython
```

Мы собираемся использовать данную аналогию, чтобы донести концепции до вас. Боб и Алиса сгенерируют свои собственные

пары открытого и закрытого ключа, а Боб зашифрует сообщение для Алисы, чтобы она могла его расшифровать. PyNaCl предоставляет нам очень полезный класс Box, который имитирует предыдущую аналогию.

Погнали:

```
from nacl.public import PrivateKey, Box #

Сгенерирем секретные ключи для Алисы и Боба
alices_private_key = PrivateKey.generate()
bobs_private_key = PrivateKey.generate()

# Публичные ключи генерируются из закрытых ключей

alices_public_key = alices_private_key.public_key
bobs_public_key = bobs_private_key.public_key

# Боб отправит Алисе сообщение...
# Поэтому он создает коробку со своим закрытым ключом и
открытым ключом Алисы
bobs_box = Box(bobs_private_key, alices_public_key)

# Мы шифруем секретное сообщение Боба (байты)...
encrypted = bobs_box.encrypt(b"I am Satoshi")

# Алиса создает второй ящик со своим закрытым ключом и
открытым ключом Боба, чтобы она могла расшифровать
сообщение
alices_box = Box(alices_private_key, bobs_public_key)

# Теперь Алиса может расшифровать
сообщение:

plaintext = alices_box.decrypt(encrypted)
print(plaintext.decode('utf-8'))

I am Satoshi
```

PyNaCl вызывает исключение, если сообщение было подделано или его не удалось расшифровать.

Цифровые подписи

Цифровые подписи существуют по многим из тех же причин, по которым вы должны подписывать что-либо в реальной жизни: они не оставляют получателю никаких сомнений в подлинности документа. Подписи удовлетворяют трем полезным требованиям:

1. Подлинность: «Это могло быть подписано только Даниэлем».
2. Целостность: «Эти данные не были подделаны или скомпроментированы».
3. Безотказность: «Дэниел не может отрицать, что отправил данные».

Цифровые подписи используют криптографию с открытым ключом, чтобы удовлетворить этим требованиям. Как и в нашем примере, где Алиса создает «ящик» с открытым ключом Боба, мы используем открытый ключ Боба, чтобы убедиться в том, что только Боб мог отправить и подписать часть данных. Идея проста: любой, у кого есть мой открытый ключ, может быстро убедиться в том, что действительно я подписал сообщение.

Вот схема данной идеи.

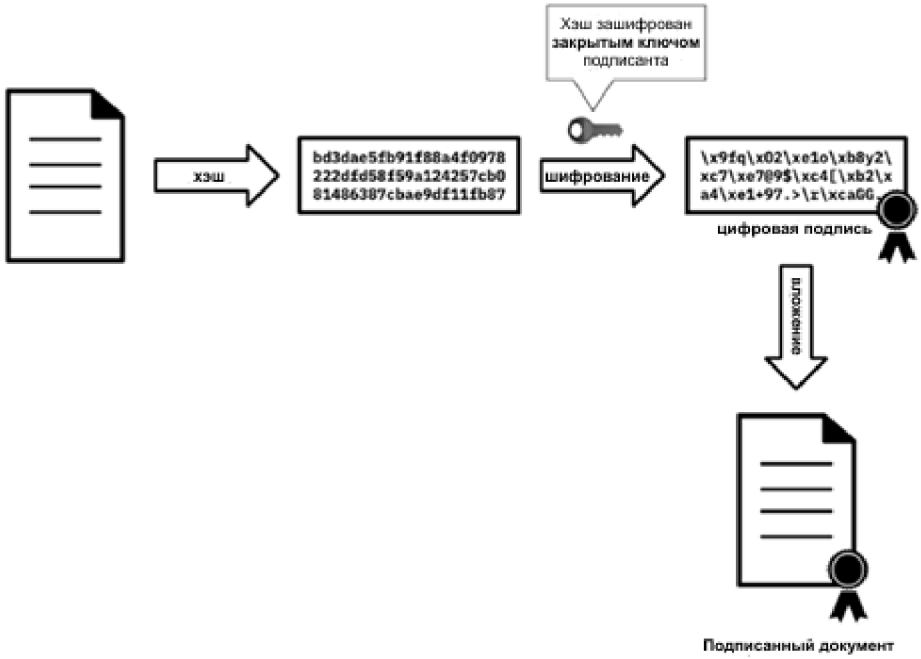


Рис. 6-2. *Цифровая подпись документа*

1. Во-первых, наши незашифрованные данные в виде открытого текста хэшируются (это предотвращает подделку).
2. Затем хэш шифруется с помощью закрытого ключа.
3. Затем мы присоединяем (конкатенируем) зашифрованный хэш к данным.

Давайте посмотрим, как это выглядит на Python. Во-первых, с точки зрения Боба, мы создадим пару ключей; затем подпишем ими сообщение. После этого мы увидим, что любой может использовать этот открытый ключ для проверки сообщения.

```
1  import nacl.encoding
2  import nacl.signing
3
4  # Generate a new key-pair for Bob
```

```
5  bobs_private_key = nacl.signing.SigningKey.generate()
6  bobs_public_key = bobs_private_key.verify_key7
8  # Поскольку это байты, нам нужно сериализовать ключ в
   читаемый формат перед его публикацией:
9  bobs_public_key_hex = bobs_public_key.encode(encoder=nacl.
   encoding.HexEncoder)
10
11        # Теперь давайте подпишем сообщение им
12        signed = bobs_private_key.sign(b"Send $37 to Alice")
```

Открытый ключ (в шестнадцатеричном формате), сгенерированный в строке 10:

```
e7ff10ede8a691b982516059a0627d369504e3633e0297e28ec5fc71994577d3
```

Во-первых, давайте посмотрим, как выглядит подписанное сообщение в строке 12. Как видите, сообщение не зашифровано! Но оно дополнено байтами, содержащими подпись:

```
b'\x9fq\x02\xe1o\xb8y2\xc7\xe7@9$\xc4[\xb2\xa4\xe1+97.>\r\
xcaGG\x8a
Y\x86\xc3\xfe\xb9W{\xc4\x9c\x87\x00(\x1d\xe9}j\xe4\xed\xd2\
x0b\xcb\x88\x87J\xecy\x04GQ
H\xea\xcc\xc2\xe7\x03Send $37 to Alice'
```

Проверка

В процессе проверки используется открытый ключ подписывающей стороны для проверки подписи.

Рис. 6-3. *Проверка цифровой подписи*

Давайте посмотрим, как мы можем использовать открытый ключ для проверки того, что сообщение было подписано Бобом:

```
import nacl.encoding
import nacl.signing


# Из приведенного выше примера...
bobs_public_key = b'e7ff10ede8a691b982516059a0627d369504e3633e0297e28ec5fc71994577d3'

# Генерируем verify_key
verify_key = nacl.signing.VerifyKey(bobs_public_key, encoder=nacl.encoding.HexEncoder)

signed_message = b'\x9fq\x02\xe1o\xb8y2\xc7\xe7@9$\xc4[\xb2\xa4\xe1+97.>\r\xca GG\x8a Y\x86\xc3\xfe\xb9W{\xc4\x9c\x87\x00(\x1d\xe9}j\xe4\xed\xd2\x0b\xcb\x88\x87 J\xecy\x04GQH\xea\xcc\xc2\xe7\x03Send $37 to Alice'

```

```
13  # Теперь мы пытаемся проверить сообщение
14  # Любая нелегитимность приведет к возникновению исключения
15      verify_key.verify(signed_message)
```

Примечание Имеет ли смысл предыдущий пример? Хорошо было бы сделать паузу и потренироваться в интерпретаторе Python, пока вы не поймете суть использования NaCl. Это важно для того, когда мы изучим следующую главу о транзакциях.

Кошельки на блокчейне

В отличие от Биткойна, Ethereum — это модель, основанная на учетных записях, что означает, что каждый «пользователь» в блокчейне будет иметь учетную запись. Биткойн не имеет понятия счета; вместо этого это система, которая очень похожа на то, как наличные деньги вводятся и выводятся из физического кошелька. Система Биткойн называется UTXO (Unspent Transaction Outputs) и представляет собой элегантную структуру данных для моделирования транзакций. Мы поговорим о UTXO в следующей главе, а пока давайте сосредоточимся на модели Ethereum, основанной на учетных записях.

Когда вы впервые взаимодействуете с Ethereum, первое, что вы обычно делаете, — это генерируете пару ключей. Ваш *адрес* Ethereum — это просто ваш открытый ключ. Ваш закрытый ключ хранится в безопасном месте, либо в каком-то программном обеспечении, либо в аппаратном кошельке. Чтобы кто-то мог отправить вам деньги через блокчейн Ethereum, ему просто нужно знать ваш открытый ключ. Но только вы можете получить доступ к этим деньгам, потому что только у вас есть закрытый ключ.

ГЛАВА 7

Создание транзакционного узла

Чтобы получить полноценную криптовалюту, данные в нашей цепочке блоков должны быть *транзакциями*. Каждая транзакция передает право собственности на монеты от одного закрытого ключа другому. Транзакции собираются в каждом блоке и *добываются*, увеличивая блокчейн; на самом деле, чем старше блок, тем больше в нем надежности — т.е. с большей вероятностью он будет частью блокчейна де-факто. В любой момент времени майнеры заняты майнингом несколько разных блоков, содержащих разные транзакции — это гонка за поиском блока — когда майнер находит блок, он транслирует его, а остальные майнеры отбрасывают свои текущие блоки (они проиграли в гонке) и переходят к следующим. Эти «отброшенные» блоки обычно называют блоками-сиротами.

В этой главе мы рассмотрим все концепции, которые мы изучили, и включим их в полноценный узел, способный работать в одноранговой сети. Затем, используя криптографию из предыдущей главы, мы поясним, как совершаются и проверяются транзакции.

Резюме по транзакциям и работе

Отход от модели UTXO Биткойн

Если вы потратили какое-то время на изучение биткойнов, то вы, вероятно, слышали о UTXO или, более формально, о выходе неизрасходованных транзакций Unspent Transaction Outputs (UTXO). Эта модель элегантна, потому что она напоминает то, как в реальном мире работают твердые наличные: здесь нет счетов: деньги — это «примечание», которое может находиться в чьем-то кошельке (открытый ключ). UTXO противопоставляются модели на основе учетной записи, которая примерно так и работает с вашим банковским счетом: вы даете кому-то номер своего счета, и теперь они могут переводить на него деньги. Некоторые криптовалюты используют эту модель, в первую очередьEthereum.

Это та модель, которую мы будем реализовывать, потому что ее легко обсуждать, тестировать и объяснять. Мы создадим класс таким образом, чтобы его можно было легко заменить моделью UTXO (оставим это читателю в качестве упражнения).

Роль майнера

Как мы видели в ГЛАВЕ 4, роль майнера состоит в том, чтобы генерировать новые монеты, находя подходящий хэш для блока. Во время майнинга майнер собирает входящие транзакции в пул, известный в биткойнах как мемпул, ожидая включения в следующий блок. Если транзакций слишком много для заполнения блока, майнер выбирает те, у которых самые высокие комиссии, чтобы увеличить свою прибыль. Во время криптовалютного бума 2017 года цена транзакций Биткойн превысила 40 долларов из-за большого количества транзакций, которые были зарезервированы. Темы масштабирования Биткойна часто включают мемпул и комиссии за транзакции, в первую очередь вопрос о размере блока:

сколько транзакций мы должны хранить в блоке? В биткойнах ограничение составляет 1 мегабайт, что в среднем составляет около 1700 транзакций.

Как мы будем реализовывать транзакции

Мы внесем некоторые изменения в наш узел funcoin; в частности, мы должны выяснить, как транзакции распространяются по нашей сети. То, как структурированы одноранговые сети, имеет решающее значение для надежности криптовалюты. Биткойн считается «пуш-сетью»: вместо того, чтобы узел запрашивал у своих пиров новые транзакции, узел ***отправляет*** новую транзакцию всем своим пирам, когда получает ее. Грубо говоря, когда новый узел присоединяется к сети, он должен создавать эффект *gossip*, благодаря которому одноранговые узлы узнают о его присутствии и могут надежно отправлять ему будущие транзакции.

Транзакция имеет простую структуру данных, содержащую

1. Открытый ключ отправителя
2. Открытый ключ получателя
3. Сумма перевода
4. Объявленная комиссия (подумайте о «чаевых», чтобы стимулировать майнеров включить транзакцию в свой следующий блок)

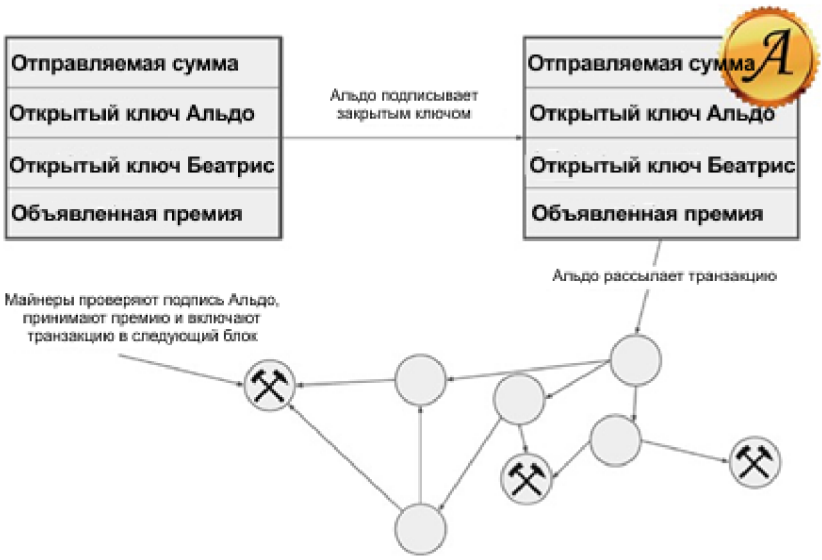


Рис. 7-1. *Создание транзакции с использованием цифровых подписей*

Чтобы отправлять или получать фанкоины, нам нужно сгенерировать кошелек — пару **открытого** и закрытого ключей. Чтобы отправить фанкойны Беатрис, Альдо должен создать транзакцию, содержащую **открытые** ключи его и Беатрис, а также сумму перевода. Затем транзакция подписывается с использованием закрытого ключа Альдо.

Любые одноранговые узлы сети могут подтвердить подлинность транзакции, проверив открытый ключ Альдо. Если транзакция проходит проверку, то позже, когда майнер включит ее, она будет добавлена в блокчейн.

Опечатки и изменения

По мере того, как меняются ресурсы и библиотеки, а люди обнаруживают ошибки в этой начальной реализации, со временем становится неизбежным, что рабочий код будет отличаться от напечатанного, надеюсь, незначительным образом. пожалуйста, держите общедоступный репозиторий github под рукой. Если вам нужно, посетите: https://github.com/dvf/learn- blockchains-book/.

Создание проекта для нашего полного узла

Установка ресурсов

На этом этапе проекта мы охватим все элементарные единицы, необходимые для построения нашего узла:

- Блокчейн для поддержания неизменной цепочки данных
- Сервер, позволяющий клиентам подключаться и отправлять данные туда и обратно
- Алгоритм майнинга для создания новых фанкойнов
- Основы криптографии, позволяющие нам проверять и создавать транзакции

Теперь нам нужно рассмотреть архитектуру нашего узла и реструктурировать наш проект, используя заданные элементарные единицы в качестве импортируемых модулей. Для этого мы создадим новый проект Python (папку). Давайте создадим новую папку и запустим ее как пустой проект Poetry:

```
$ mkdir funcoin
$ cd funcoin
$ poetry init -n
```

И добавьте необходимые ресурсы (через минуту я расскажу, для чего нужен каждый из них):

```
$ poetry add pynacl structlog colorama marshmallow marshmallow-oneofschema aiohttp
```

Примечание Вы можете заметить добавление *structlog* — позже мы будем использовать его вместо операторов `print()` которыми мы замусорили наш код, чтобы получить стройный, осмысленный вывод.

Poetry теперь создаст виртуальную среду в соответствующем месте и установит ресурсы:

```
Creating virtualenv funcoin-PPSjSr3P-py3.8 in
/Users/dvf/Library/Caches/pypoetry/virtualenvs
Using version ^20.1.0 for structlog
Using version ^1.3.0 for pynacl
Updating dependencies
Resolving dependencies...

(1.2s)Writing lock file

Package operations: 22 installs, 0 updates, 0 removals

  - Installing idna (2.9)
  - Installing multidict (4.7.6)
  - Installing pycparser (2.20)
  - Installing pyparsing (2.4.7)
  - Installing six (1.14.0)
  - Installing async-timeout (3.0.1)
  - Installing attrs (19.3.0)
  - Installing cffi (1.14.0)
  - Installing chardet (3.0.4)
  - Installing colorama (0.4.3)
  - Installing marshmallow (3.6.0)
  - Installing more-itertools (8.3.0)
  - Installing packaging (20.3)
  - Installing pluggy (0.13.1)
  - Installing py (1.8.1)
```

```
- Installing wcwidth (0.1.9)
- Installing yarl (1.4.2)
- Installing aiohttp (3.6.2)
- Installing marshmallow-oneofschema (2.0.1)
- Installing pynacl (1.3.0)
- Installing pytest (5.4.2)
- Installing structlog (20.1.0)
```

Теперь у вас есть файлы pyproject.toml и poetry.lock files в вашем проекте funcoin. Вот краткое изложение того, для чего мы будем использовать каждый пакет:

Пакет	Описание
Pynacl	подписание и проверка транзакций
Structlog	библиотека протоколирования (лучше, чем операторы print())
Colorama	позволяет вывод в цвете (в журнале)
Marshmallow	проверяет структуры данных, такие как сообщения Json, которые наш узел будет отправлять и получать
marshmallow-oneofschema	позволяет marshmallow проверять более сложные структуры данных
Aiohttp	асинхронный http-клиент (он понадобится нам, чтобы найти наш общедоступный IP-адрес)

Создание файловой структуры

Теперь мы собираемся создать вложенную папку funcoin для размещения наших «элементарных» единиц — различных аспектов или классов, которые при объединении дадут нам полный узел.

Создайте слдющие пустые файлы:

```
touch node.py
mkdir funcoin
cd funcoin
touch __init__.py
touch blockchain.py
touch connections.py
touch transactions.py
touch server.py
touch types.py
touch messages.py
touch utils.py
```

В конечном виде ваша папка должна выглядеть так:

```
funcoin
├── funcoin
│   ├── __init__.py
│   ├── blockchain.py
│   ├── connections.py
│   ├── messages.py
│   ├── peers.py
│   ├── server.py
│   ├── transactions.py
│   ├── types.py
│   └── utils.py
├── __init__.py
├── node.py
├── poetry.lock
└── pyproject.toml
```

Пустые файлы внутри `funcoin/funcoin` будут содержать наш реструктурированный код из предыдущих глав. Давайте рассмотрим структуру нашего узла.

Структурирование нашего узла

Делегирование обязанностей

Мы собираемся использовать node.py в качестве точки входа для запуска нашего узла. Вы спрашиваете «Что такое точка входа?». Думайте об этом понятии, как о каноническом способе запуска нашего узла.

Я считаю, что при разработке программ полезно быть как можно более абстрактными — прежде чем что-либо реализовывать, спроектируйте то, как вы хотите, чтобы программа работала:

```
# Instantiate the server with some config
server = Server(**some_config)

# Start the server
server.run()
```

Остановитесь здесь на мгновение и подумайте о различных фрагментах, которые мы будем реализовывать: как вы думаете, должен ли наш поддельный класс Server() отвечать за хранение нашей цепочки блоков? Как насчет совершения сделок? Если он отвечает за запуск *всего*, то как мы сможем заменить нашу модель транзакций, скажем, на UTXO на более позднем этапе?

Здравый принцип разработки приложений заключается в том, что *каждый модуль должен выполнять одну задачу и делать ее хорошо*. Вот лучший способ абстрагировать наш сервер:

```
from funcoin.blockchain import Blockchain
from funcoin.pool import ConnectionPool
from funcoin.server import Server

# Создаем блокчейн и наш пул «пиров»

blockchain = Blockchain()
```

```
connection_pool = ConnectionPool()

# Создаем экземпляр сервера и «прикручиваем» наши модули

server = Server(blockchain, connection_pool)

# Now start the server
server.start()
```

Теперь у нас гораздо более чистая абстракция — нашему серверу вообще не нужно заниматься блокчейном; ему просто нужно беспокоиться о том, чтобы быть сервером. Не объясняя сейчас различные модули, вот как выглядит наш окончательный файл node.py.

Листинг 7-1. funcoin/node.py

```
1 import asyncio2
3 from funcoin.blockchain import Blockchain
4 from funcoin.connections import ConnectionPool
5 from funcoin.peers import P2PProtocol
6 from funcoin.server import Server7
8 blockchain = Blockchain() ①
9 connection_pool = ConnectionPool() ②
10
11  server = Server(blockchain, connection_pool,
    P2PProtocol)12
13
14      async def main():
15   # Start the server
16   await server.listen()
17
18
19      asyncio.run(main())
```

Обратите внимание на создание экземпляров классов Blockchain ①

и ConnectionPool ②. Это гарантирует, что они воссоздают синглтоны с отслеживанием состояния, а это означает, что любой модуль, выполняющий импорт из узла, connection_pool будет импортировать один и тот же созданный объект, тем самым сохраняя состояние.

Давайте установим цель наших четырех критических модулей, прежде чем внедрять их.

Модуль	Описание
blockchain	Модуль для размещения нашего класса Blockchain, содержащего де-факто блокчейн, который мы создали в предыдущих главах
Peers	Логика обработки распространения сообщений, которые могут отправлять нам одноранговые узлы: как мы поддерживаем связь. Мы назовем это P2PProtocol
connections	Логика для обработки «пула» активных соединений, взаимодействующих с нашим узлом
server	Где расположен наш базовый TCP-сервер

Примечание Поскольку мы используем asyncio (в отличие от потоков), мы значительно упрощаем проблему параллелизма: при использовании потоков программисту нужно беспокоиться об условиях конкуренции (два или более потока конкурируют за обновление или получение чего-либо). Обратная сторона этого подход заключается в том, что код может стать трудным для чтения и осмысления (но, по моему опыту, оно того стоит, если вы потратите время, чтобы следовать ему).

Наша точка входа имеет единственную функцию main(). Типичная для асинхронной программы, она используется для раскрутки нашего сервера.

Серверный модуль

Мы преобразуем чат-сервер, созданный в ГЛАВЕ 5, и адаптируем его для наших целей в файле funcoin/server. py.

Вот примерное описание нашего класса.

Листинг 7-2. server.py

```
class Server:
 def_init (self, blockchain, connection_pool, p2p_protocol):
...
async def get_external_ip(self):
# Находит наш «внешний IP», чтобы мы могли объявить его
нашим коллегам
...

async def handle_connection(self, reader: StreamReader,
writer: StreamWriter):
# Эта функция вызывается, когда мы получаем новое
соединение
# Объект `write` представляет подключающийся пир

while True:try:
# Мы обрабатываем и/или отвечаем на входящие данные
...

except (asyncio. exceptions. IncompleteReadError,
ConnectionError):
# Произошла ошибка, выход из цикла ожидания

async def listen(self, hostname="0. 0. 0. 0", port=8888):
# Это метод прослушивания, который порождает наш сервер

server = await asyncio. start_server(self. handle_
connection, hostname, port)
        logger. info(f"Server listening on {hostname}:{port}")
```

```
        async with server:
await server.serve_forever()
async def get_external_ip(self):
# Находит наш «внешний IP», чтобы мы могли объявить его
нашим коллегам
...

async def handle_connection(self, reader: StreamReader,
writer: StreamWriter):
# Эта функция вызывается, когда мы получаем новое
соединение
# Объект `write` представляет подключающийся пир

while True:try:
# Мы обрабатываем и/или отвечаем на входящие данные
...

except            (asyncio.exceptions.IncompleteReadError,
ConnectionError):
# Произошла ошибка, выход из цикла ожидания

async def listen(self, hostname="0.0.0.0", port=8888):
# Это метод прослушивания, который порождает наш сервер

server = await asyncio.start_server(self.handle_
connection, hostname, port)
        logger.info(f"Server listening on {hostname}:{port}")

        async with server:
            await server.serve_forever()
```

Разберем методы последовательно. Не пугайтесь, если при первом просмотре вы не «поймёте»; мы будем часто возвращаться к серверу, поскольку продолжим загружать наши модули.

Листинг 7-3. funcoin/node.py

```
1  import asyncio
2  from asyncio import StreamReader, StreamWriter
3
4  import structlog
5  from marshmallow.exceptions import MarshmallowError
6
7  from funcoin.messages import BaseSchema
8  from funcoin.utils import get_external_ip
9
10 logger = structlog.getLogger() ⑦
11
12
13 class Server:
14     def   init__(self, blockchain,
           connection_pool,p2p_protocol):
15     self.blockchain = blockchain ①
16           self.connection_pool = connection_pool
17           self.p2p_protocol = p2p_protocol
18           self.external_ip = None
19           self.external_port = None
20
21           if not (blockchain and 22 connection_pool and
           p2p_protocol):
22            logger.error("'blockchain', 'connection_pool', and
           'gossip_protocol' must all be instantiated")
23            raise Exception("Could not start")
24
25  async def get_external_ip(self):
26      self.external_ip = await get_external_ip() ②
27
28  async def handle_connection(self, reader: StreamReader,
    writer: StreamWriter):
29      while True:
```

```
30          try:
31              # Ждать, пока не поступят новые данные
32              data = await reader.readuntil(b"\n") ③
33
34              decoded_data = data.decode("utf8").strip() ④
35
36              try:
37                  message = BaseSchema().loads(decoded_
                    data) ⑤
38              except  MarshmallowError:
39                  logger.info("Received unreadable
                    message", peer=writer)
40                  break
41
42              # Извлекаем адрес из сообщения, добавляем его
                  в объект записи
43              writer.address = message["meta"]["address"]
44
45              # Давайте добавим пир в наш пул соединений
46              self.connection_pool.add_peer(writer)
47
48                  #. . . и обрабатаем сообщение
49                  await self.p2p_protocol.handle_
                    message(message, writer) ⑥
50
51                  await writer.drain()
52                  if writer.is_closing():
53                      break
54
55          except (asyncio.exceptions.IncompleteReadError,
            ConnectionError):
56              # Произошла ошибка, выйти из цикла ожидания
57              break
```

```
58
59          # Соединение закрыто. Давайте сбросим...
60          writer.close()
61          await  writer.wait_closed()
62          self.connection_pool.remove_peer(writer)⑦
63
64      async def listen(self, hostname="0.0.0.0", port=8888):
65          server = await asyncio.start_server(self.handle_
            connection, hostname, port)
66          logger.info(f"Server listening on {hostname}:{port}")
67
68          self.external_ip = await self.get_external_ip()
69          self.external_port = 8888
70
71          async with server:
72              await server.serve_forever()
```

① Вот как мы «загружаем» наши модули на сервер: класс сервера (и все, что к нему подключено) всегда будет иметь доступ к нашему блокчейну посредством `self.blockchain`.

② Хотя мы еще не реализовали функцию `get_external_ip()`, она отвечает за поиск нашего внешнего IP-адреса.

③ Здесь мы ждем ***бесконечно***, пока нам не будет отправлено сообщение, заканчивающееся символом новой строки (\n) character. Это первая из потенциальных уязвимостей, которой вам следует остерегаться, так как кто угодно может просто спамить наш сервер бесконечным сообщением.

④ Мы пытаемся декодировать сообщение, предполагая, что оно было отправлено в виде строки в формате UTF-8.

⑤ Возможно, самым большим сюрпризом, о котором мы скоро узнаем, является использование marshmallow (библиотека, которую мы установили ранее) для разбора и проверки входящего сообщения от однорангового узла.

⑥ Как только сообщение будет успешно проанализировано, дальнейший код может предположить, что все соответствующие поля существуют, и мы можем использовать наш протокол p2p, чтобы выяснить, что делать.

⑦ Обратите внимание на импорт и использование *structlog* — мы используем его для замены операторов print(). Это дает хорошо читаемый вывод на консоль, когда мы запускаем весь наш узел. Например, он сообщает вам, из какого файла был получен журнал.

Неважно, что вы понимаете мельчайшие детали того, что мы реализовали в классе Server. Следуйте общей логике, принимая во внимание более важные принципы.

Блокчейн-модуль

Давайте вернемся к классу Blockchain, который мы создали в ГЛАВЕ 3, и вставим его в funcoin/blockchain.py.

Листинг 7-4. funcoin/blockchain.py

```
import asyncio
import json
import math
import random
from hashlib import sha256
from time import time

```

```
 8  import structlog
 9
10  logger = structlog.getLogger("blockchain")
11
12
13  class Blockchain(object):
14      def __init__ (self):
15          self.chain = []
16          self.pending_transactions = []
17          self.target = "0000ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff"
18
19          # Создаем блок генезиса
20          logger.info("Creating genesis block")
21          self.chain.append(self.new_block())
22
23      def new_block(self):
24              block = self.create_block(
25               height=len(self.chain),
26               transactions=self.pending_transactions,
27               previous_hash=self.last_block["hash"] if
                 self.last_block else None,
28                   nonce=format(random.getrandbits(64), "x"),
29                   target=self.target,
30                   timestamp=time(),
31              )
32
33          # Сбрасываем список незавершенных транзакций
34              self.pending_transactions = []
35
36              return block
37
38      @staticmethod
```

```
def create_block(
    height, transactions, previous_hash, nonce,
    target, timestamp=None
):
        block = {
            "height": height,
            "transactions": transactions,
            "previous_hash": previous_hash,
            "nonce": nonce,
            "target": target,
            "timestamp": timestamp or time(),
        }

        # Получаем хэш этого нового блока и добавляем его в
          блок
        block_string = json.dumps(block, sort_keys=True).
        encode()
        block["hash"] = sha256(block_string).hexdigest()
        return block

    @staticmethod
    def hash(block):
        # Мы гарантируем, что словарь отсортирован, иначе у
          нас будут несогласованные хэши
        block_string = json.dumps(block, sort_keys=True).
        encode()
        return sha256(block_string).hexdigest()

    @property
    def last_block(self):
        # Возвращает последний блок в цепочке (если есть
          блоки)
        return self.chain[-1] if self.chain else None
```

```
66
67      def valid_block(self, block):
68          # Проверяем, меньше ли хэш блока, чем
            заданный. . .
69          return block["hash"] < self.target
70
71      def add_block(self,  block):
72          # TODO: Добавьте правильную логику проверки
            здесь!
73          self.chain.append(block)
74
75  def recalculate_target(self, block_index):
76      """
77      Возвращает число, которое нам нужно получить ниже
        заданного, чтобы добыть блок
78      """
79      # Проверяем, нужно ли нам пересчитывать заданное
80      if block_index % 10 == 0:
81          # Ожидаемый промежуток времени 10 блоков
82          expected_timespan = 10 * 10
83
84          # Рассчитываем фактический промежуток времени
85          actual_timespan = self.chain[-1]["timestamp"] -
            self.chain[-10]["timestamp"]
86
87          # Выясняем смещение
88          ratio = actual_timespan / expected_timespan
89
90          # Теперь давайте настроим соотношение так, чтобы оно
            не было слишком экстремальным
91          ratio = max(0.25, ratio)
92          ratio = min(4.00, ratio)
93
94          # Рассчитайте новое заданное значение, умножив
```

```
            текущее на коэффициент
95          new_target = int(self.target, 16) * ratio
96
97          self.target = format(math.floor(new_target),
            "x").zfill(64)
98          logger.info(f"Calculated new mining target:
            {self.target}")
99
100         return self.target
101
102     async def get_blocks_after_timestamp(self, timestamp):
103         for index, block in enumerate(self.chain):
104             if timestamp < block["timestamp"]:
105                 return self.chain[index:]
106
107     async def mine_new_block(self):
108         self.recalculate_target(self.last_block["index"] + 1)
109         while True:
110             new_block = self.new_block()
111             if self.valid_block(new_block):
112                 break
113
114                 await asyncio.sleep(0)
115
116         self.chain.append(new_block)
117         logger.info("Found a new block: ", new_block)
```

Единственное изменение здесь — добавление `structlog`, где мы заменили каждый оператор `print()` полезным регистратором.

Модуль соединений

В `funcoin/connections.py` мы собираемся хранить всю логику,

которая управляет нашим ConnectionPool нашего чат-сервера из ГЛАВЫ 5. Вот схема класса:

```
class ConnectionPool:
  def __init__(self):
  ...
  def broadcast(self,      message):
      # Способ передачи сообщения всем подключенным одноранговым
      узлам
      ...

 @staticmethod
 def get_address_string(writer):
      # Получить ip:port (адрес) пира
      ...

 def add_peer(self, writer):
      # Добавим пир в наш пул соединений
      ...

 def remove_peer(self, writer):
      # Удаляем одноранговый узел из нашего пула соединений
      ...

 def get_alive_peers(self, count):

      # Возвращает несколько подключенных пиров
      ...
```

Давайте посмотрим, как выглядит конкретная реализация. Так как большинство этих методов уже были конкретизированы в ГЛАВЕ 5, здесь не должно быть никаких сюрпризов. Основное изменение — дополнительные методы: get_alive_ peers() и get_address_string().

Листинг 7-5. funcoin/blockchain.py

```
1  import structlog
```

```
2  from more_itertools import take3
4  logger = structlog.getLogger(__name__)
5
6
7  class ConnectionPool:
8    def __init__ (self):
9          self.connection_pool = dict() ①
10
11     def broadcast(self,  message):
12         for user in self.connection_pool:
13             user.write(f"{message}".encode())
14
15     @staticmethod
16     def get_address_string(writer):
17         ip = writer.address["ip"]
18         port = writer.address["port"]
19         return f"{ip}:{port}" ②
20
21     def add_peer(self, writer):
22         address = self.get_address_string(writer)
23         self.connection_pool[address] = writer
24         logger.info("Added new peer to pool",
           address=address)
25
26     def remove_peer(self, writer):
27         address = self.get_address_string(writer)
28         self.connection_pool.pop(address)
29         logger.info("Removed peer from pool",
           address=address)
30
31     def get_alive_peers(self, count):
32         # TODO (Reader): Отсортируем их по наиболее
             активным, но пока давайте просто получим первое
```

```
                *количество* из них
33              return take(count, self.connection_pool.items()) ③
```

① Здесь мы используем dict, сопоставляющий адрес для writer (представляющего одноранговое соединение).

② Строка address в сопоставлении — это просто комбинация ip:port одноранговой узла — это важно, потому что именно так мы будем однозначно идентифицировать соединения.

③ Мы используем функцию take() для возврата количества пиров count из нашего пула.

Модуль пиров

Это, пожалуй, на данный момент самый важный модуль: он представляет логику, связанную с отправкой и получением сообщений в нашей одноранговой сети.

Давайте введем методы в funcoin/peers.py; они будут удивительно простыми:

```
class P2PError(Exception): ①
    pass

class P2PProtocol:
     def __init__(self, server):

     ...
    @staticmethod
    async def send_message(writer, message):
        # Отправляет сообщение определенному одноранговому
          узлу (объект записи)
        ...

    async def handle_message(self, message, writer):
```

```
        # Обрабатывает входящее сообщение, переданное
        сервером
        # Передает это сообщение более конкретному методу:
          handle_<method name>()
            ...

    async def handle_ping(self, message, writer):
            # Обрабатывает входящее сообщение "ping"
            ...

    async def handle_block(self, message, writer):
            # Обрабатывает входящее «block» сообщение
            ...

    async def handle_transaction(self, message, writer):
            # Ручки во входящем сообщении "transaction"
            ...

    async def handle_peers(self, message, writer):
            # Обрабатывает входящие сообщения "peers"
            ...
```

① Нам понадобится класс ошибок, который мы можем использовать для «отлова» проблем в коде импортирования.

Комментарии в каждом методе объясняют код, который нам нужно реализовать. Наиболее важным методом здесь является метод handle_message(). Мы еще не определили сообщения, которые может отправлять каждый одноранговый узел, поэтому давайте определим базовую форму сообщения:

```
{
   "meta": {
     "address": {
       "ip": <external ip: str>,
       "port": <external port: int>
```

```
        },
        "client": "funcoin 0.1"
      },
    "message": {
      "name": <message name: str>,
      "payload": <message payload: object>
    }
}
```

То есть *все* сообщения, отправляемые в нашей одноранговой сети, имеют такую структуру. Мета-ключ содержит информацию об одноранговом узле, отправившем сообщение (даже если этот одноранговый узел — мы), а ключ сообщения содержит имя и полезную информацию отправляемого сообщения. Для funcoin мы будем реализовывать четыре сообщения.

Сообщение	Описание
Ping	Начальное сообщение, которое узел отправляет одноранговому узлу при инициировании соединения
Transaction	Одна транзакция, которая распространяется на всем протяжении от одного однорангового узла до другого
Peers	Куча адресов, о которых одноранговый узел может знать или не знать (для построения своей сети)
Block	Один блок (возможно, недавно добытый) для однорангового узла для его добавления в свою цепочку блоков

Прежде чем мы подробно поговорим о том, как выглядят эти сообщения, давайте завершим модуль funcoin/peers.py, конкретизировав данные методы.

Листинг 7-6. funcoin/peers.py

```
import asyncio2
import structlog
from funcoin.messages import (
 create_peers_message,
  create_block_message,
  create_transaction_message,
create_ping_message,9
)
from funcoin.transactions import validate_transaction

logger = structlog.getLogger(__name__)


class P2PError(Exception):
    pass


class P2PProtocol:
    def __init__ (self, server):
        self.server = server
        self.blockchain = server.blockchain
        self.connection_pool = server.connection_pool

    @staticmethod
    async def send_message(writer, message):
        writer.write(message.encode() + b"\n")

    async def handle_message(self, message, writer):
        message_handlers = {
            "block": self.handle_block,
            "ping": self.handle_ping,
```

```
33              "peers": self.handle_peers,
34          "transaction": self.handle_transaction, 35}
36
37          handler = message_handlers.get(message["name"])
38      if not handler:
39          raise P2PError("Missing handler for message")
40
41      await handler(message, writer)
42
43  async def handle_ping(self, message, writer):
44      block_height = message["payload"]["block_height"]
45
46      # Если они майнеры, давайте отметим их как таковых
47      writer.is_miner = message["payload"]["is_miner"]
48
49      # Отправим этому пользователю 20 самых «живых» пиров
50      peers = self.connection_pool.get_alive_peers(20)
51      peers_message = create_peers_message(self.server.
        external_ip, self.server.external_port, peers
52      await self.send_message(writer, peers_message)
53
54      # Давайте отправим им блоки, если их у них меньше, чем у
        нас
55      if block_height < self.blockchain.last_block
        ["height"]:
56          # Отправим им каждый блок последовательно,
              с их высоты
57          for block in self.blockchain.chain
            [block_height + 1:]:
51              await self.send_message(
52                  writer,
53                  create_block_message(
54                      self.server.external_ip, self.
```

```
                        server.external_port, block
55                  ),
56              )
57
58      async def handle_transaction(self, message, writer):
59          """
60          Выполняется, когда мы получаем транзакцию,
            которая была передана пиром
61          """
62          logger.info("Received transaction")
63
64          # Подтверждает транзакцию
65          tx = message["payload"]
66
67          if validate_transaction(tx) is True:
68              # Добавляем tx в наш пул и распространяем его
                среди наших пиров
69              if tx not in self.blockchain.pending_transactions:
70                  self.blockchain.pending_transactions.append(tx)
71
72                  for peer in self.connection_pool.get_alive_
                    peers(20):
73                      await self.send_message(
74                          peer,
75                          create_transaction_message(
76                              self.server.external_ip, self.
                                server.external_port, tx
77                          ),
78                      )
79          else:
80              logger.warning("Received invalid transaction")
81
82      async def handle_block(self, message, writer):
83          """
```

```
84          Выполняется, когда мы получаем блок, который был
            передан узлом
85          """
86      logger.info("Received new block")
87
88      block = message["payload"]
89
90      # Передаем блок в блокчейн для добавления, если он
        действителен
91      self.blockchain.add_block(block)
92
93      # Передаем блок нашим коллегам
94          for peer in self.connection_pool.get_alive_peers(20):
95              await self.send_message(
96                  peer,
97                  create_block_message(
98                   self.server.external_ip, self.server.
                     external_port, block
99                  ),
100             )
101
102 async def handle_peers(self, message,
    writer):
103     """
104     Выполняется, когда мы получаем блок, который был
        передан узлом
105     """
106     logger.info("Received new peers")
107
108     peers = message["payload"]
109
110     # Создаем ping-сообщение, которое мы отправим каждому
        узлу
111     ping_message = create_ping_message(
```

```
112             self.server.external_ip,
113             self.server.external_port,
114             len(self.blockchain.chain),
115                 len(self.connection_pool.get_alive_peers(50)),
116                 False,
117             )
118
119         for peer in peers:
120                 # Создаем подключение и добавляем их в наш
                    пул соединений в случае успеха
121                 reader, writer = await asyncio.open_connection
                    (peer["ip"], peer["port"])
122
123                 # Нас интересует только "writer"
124                 self.connection_pool.add_peer(writer)
125
126                 # Отправляем узлу сообщение PING
127                 await self.send_message(writer, ping_message)
```

Обратите особое внимание на метод handle_ping (я добавил комментарии, чтобы помочь с объяснением). Но на данный момент стоит обсудить структуру сообщений, отправляемых в нашей криптовалюте.

Обмен сообщениями

Ранее мы показали пример общей формы сообщения в формате JSON; вот как это выглядит в виде таблицы:

Таблица 7-1. *Общая форма сообщения*

Поле	Тип	Описание
Meta	мета	—
Message	сообщение	—

Метаобъект включается в каждое сообщение и содержит информацию, относящуюся к отправителю.

Таблица 7-2. *мета объект*

Поле	Тип	Описание
address	Пир	Содержит информацию о сетевом адресе узла, отправившего это сообщение
client	Строка	Наименование (версии) клиента, отправившего сообщение. Например, funcoin-0.1

Тип одноранговой узла также довольно общий. Он появляется в списках отправляемых одноранговых узлов, например, новый узел, присоединяющийся к сети, или в метаблоке каждого отправляемого сообщения.

Таблица 7-3. *тип пира (представляет одноранговый узел)*

Поле	Тип	Описание
Ip	Строка	Публичный IP пира
Port	Целое	Порт, который слушает пир
last_seen	Целое	Временная метка UTC времени последнего посещения одноранговой узла

Теперь давайте посмотрим на структуры четырех типов сообщений в нашей сети. Параметры сообщения просты; каждое сообщение имеет имя и полезную нагрузку (меняется в зависимости от типа отправляемого сообщения):

Таблица 7-4. объект сообщения

Поле	Тип	Описание
Name	Строка	Наименование сообщения (чтобы получатель знал, как его разобрать)
Payload	Объект	Полезная информация сообщения, соответствующая типу сообщения. См. в следующем разделе о различной полезной информации

Давайте определим различные виды полезной информации.

Таблица 7-5. Полезная информация ping

Ключ	Значение		
Name	Pong		
Payload	**Ключ**	**Тип**	**Пример**
	block_height	Целое	2000
	peer_count	Целое	23
	is_miner	Булево	Ложь

Таблица 7-6. Полезная информация транзакции

Ключ	Значение		
Name	Транзакция		
Payload	**Ключ**	**Тип**	**Пример**
	Hash	строка	<sha 256 hash of this payload>
	sender	строка	<pub key of sender>
	signature	строка	<digital signature key of sender>
	timestamp	строка	1589135911
	Receiver	строка	<pub key of receiver>
	Amount	целое	<amount of funcoins to send>

Таблица 7-7. Полезная информация пиров

Ключ	Значение	
Name	Peers	
Payload	**Ключ**	**Тип**
	Peers	Список[пиры]

Таблица 7-8. Полезная информация блока

Ключ	Значение	
Name	block	
Payload	**Ключ**	**Тип**
	mined_by	строка
	transactions	Список[транзакции]
	height	целое
	difficulty	целое
	hash	строка
	previous_hash	строка
	nonce	строка
	timestamp	целое

Использование Marshmallow для проверки наших сообщений

Теперь мы будем использовать библиотеку marshmallow для проверки наших сообщений. Наш сервер выполняет два действия: чтение сообщений и отправка сообщений. Когда сервер получает сообщение, это сообщение поступает в виде строки JSON. Затем она должна быть преобразована в словарь Python и проверена; этот процесс называется десериализацией, поскольку мы переходим от сериализованной формы (JSON) к собственному словарю Python. Сериализация работает наоборот: переход от словаря к строке JSON. В итоге

1. Сервер читает сообщения, отправляемые пирами (десериализация: json → dict).
2. Сервер отправляет сообщения пирам (сериализация:

dict ⟶ json).

Для каждого объекта, который мы хотим проверить, Marshmallow требует, чтобы мы определили *схему*. Вот как выглядит схема для однорангового объекта:

```
from marshmallow import Schema, fields

class Peer(Schema):
    ip = fields.Str(required=True)
    port = fields.Int(required=True)
    last_seen = fields.Int(missing=lambda: int(time()))
```

Обратите внимание, что мы указываем тип каждого поля. Когда Marshmallow получает объект JSON, он выдает ошибку, если одно из полей не соответствует указанному типу, что экономит нам массу времени и усилий. Вот пример сериализованной строки JSON, отправленной на наш сервер:

```
payload = '{'ip': '192.168.0.1', 'port': 8888, 'last_seen':
1589780748}'
```

Теперь мы можем попросить Marshmallow десериализовать его:

```
result = Peer().loads(payload)
# <dict> {'last_seen': 1589780748, 'ip': '192.168.0.1',
'port': 8888}
```

Если одно из полей было неправильным или не прошло проверку, Marshmallow обращается к `ValidationError`.

Попробуем сериализовать результат:

```
serialized = Peer().dumps(some_payload)
# '{'last_seen': 1589780748, 'ip': '192.168.0.1', 'port': 8888}'
```

Если вы все еще не знаете, как работает Marshmallow или почему он избавляет нас от многих хлопот, стоит ознакомиться с документацией, поскольку на веб-сайте есть множество примеров: https://marshmallow.readthedocs.io

Примечание общей темой в большинстве библиотек сериализации является использование загрузок и дампов в качестве наименований методов: загрузки десериализуются из строки; загрузка десериализуется из объекта; дамп сериализуется в строку; дамп сериализуется в объект.

Реализация и проверка типов

Давайте создадим три новых файла, funcoin/types.py, funcoin/messages.py и funcoin/utils.py для размещения наших типов, сообщений и дополнительных вспомогательных функций соответственно. Ваша структура папок должна выглядеть так:

```
.
├── funcoin
│   ├── __init__.py
│   ├── blockchain.py
│   ├── connections.py
│   ├── factories.py
│   ├── messages.py
│   ├── peers.py
│   ├── server.py
│   ├── transactions.py
│   ├── types.py
│   └── utils.py
├── _init__.py
├── node.py
├── poetry.lock
└── pyproject.toml
```

Давайте сначала определим схему Marshmallow в funcoin/schema.py.

Листинг 7-7. funcoin/schema.py

```
import json
from time import time

from marshmallow import Schema, fields, validates_schema,
ValidationError


class Transaction(Schema):
    timestamp = fields.Int()
    sender = fields.Str()
    receiver = fields.Str()
    amount = fields.Int()
    signature = fields.Str()

    class Meta:
        ordered = True


class Block(Schema):
    mined_by = fields.Str(required=False)
    transactions = fields.Nested(Transaction(), many=True)
    height = fields.Int(required=True)
    target = fields.Str(required=True)
    hash = fields.Str(required=True)
    previous_hash = fields.Str(required=True)
    nonce = fields.Str(required=True)
    timestamp = fields.Int(required=True)

    class Meta:
      ordered = True

    @validates_schema
```

```
32      def validate_hash(self, data, **kwargs):
33          block = data.copy()
34          block.pop("hash")
35
36      if data["hash"] != json.dumps(block,keys=True):
37      raise ValidationError("Fraudulent block: hashis wrong")
38
39
40      class Peer(Schema):
41              ip = fields.Str(required=True)
42              port = fields.Int(required=True)
43              last_seen = fields.Int(missing=lambda: int(time()))
```

В коде мы реализуем Peer, Block и Transaction.

В Transaction мы включаем метод проверки, который называется validate_signature. Он дополнен @validates_schema — специальным признаком, указывающим, что Marshmallow должен запускать эту функцию как часть процесса проверки. Здесь мы будем проверять транзакцию: в момент десериализации, чтобы убедиться, что *любая* десериализуемая транзакция *всегда* легитимна.

В блоке мы также реализуем метод проверки, чтобы гарантировать, что любой блок, содержащийся в *любом* сообщении, всегда является легитимным.

Определение сообщений (и их схемы)

В funcoin/messages.py мы реализуем различные сообщения и их схемы, а затем протестируем их, чтобы убедиться, что они непротиворечивы. Вот классы, которые мы будем реализовывать:

- PeersMessage
- BlockMessage
- TransactionMessage

- PingPayload
- PingMessage
- Base

Обратите внимание, что полезной информацией для PeersMessage является список Peer; полезной информацией для BlockMessage является Block, а полезной информацией для TransactionMessage является Transaction. Вот почему мы сначала определили их в types. py first.

В качестве упражнения вы можете пропустить этот шаг и попробовать определить схему для каждого сообщения самостоятельно.

Вот как они должны выглядеть, когда вы закончите.

Листинг 7-8. funcoin/messages.py

```
from marshmallow import Schema, fields, post_load
from marshmallow_oneofschema import OneOfSchema

from funcoin.schema import Peer, Block, Transaction, Ping


class PeersMessage(Schema):
    payload = fields.Nested(Peer(many=True))

    @post_load
     def add_name(self, data, **kwargs):
         data["name"] = "peers"
         return data


 class BlockMessage(Schema):
 payload = fields.Nested(Block)

```

```
19          @post_load
20      def add_name(self, data, **kwargs):
21      data["name"] = "block"
22      return data
23
24
25   class TransactionMessage(Schema):
26   payload = fields.Nested(Transaction)
27
28  @post_load
29  def add_name(self, data, **kwargs):
30      data["name"] = "transaction"
31      return data
32
33
34      class PingMessage(Schema):
35  payload = fields.Nested(Ping)
36
37  @post_load
38  def add_name(self, data, **kwargs):
39      data["name"] = "ping"
40      return data
41
42
43  class MessageDisambiguation(OneOfSchema):
44      type_field = "name"
45      type_schemas = {
46          "ping": PingMessage,
47          "peers": PeersMessage,
48          "block": BlockMessage,
49      "transaction": TransactionMessage,50
        }
51      def get_obj_type(self, obj):
```

```
52          if isinstance(obj, dict):
53              return obj.get("name")
54
55  class MetaSchema(Schema):
56      address = fields.Nested(Peer())
57      client = fields.Str()
58
59
60  class BaseSchema(Schema):
61      meta = fields.Nested(MetaSchema())
62      message = fields.Nested(MessageDisambiguation())
63
64
65  def meta(ip, port, version="funcoin-0.1"):
66      return {
67          "client": version,
68          "address": {
69              "ip": ip,
70          "port": port71
            },
72      }
73
74
75      def
        create_peers_message(external
        _ip,external_port, peers):
76      return BaseSchema().dumps({
77       "meta": meta(external_ip, external_port),
78       "message": {
79           "name": "peers",
80       "payload":      peers
81       }
82  }
```

```
83  )
84
85
86      def create_block_message(external_ip, external_port,
        block):
87      return BaseSchema().dumps({
88       "meta": meta(external_ip, external_port),
89       "message": {
90           "name": "block",
91       "payload": block92
         }
93  }
94  )
95
96   def create_transaction_message(external_ip, external_port, tx):
97  return BaseSchema().dumps(
98  {
99          "meta": meta(external_ip, external_port),
100         "message": {
101             "name": "transaction",
102         "payload": tx,103
            },
104     }
105 )
```

Обратите внимание, что мы также включаем некоторые вспомогательные функции для создания сообщений, например, в строке 89 мы включаем create_block_message, которая создает сообщение, содержащее блок. Давайте проверим это на блоке:

```
some_block = {
    "mined_by": "some public key",
    "transactions":   [],
    "height": 123,
```

```
    "difficulty": 10,
    "hash": "some fake hash", "previous_hash":  0,
    "nonce":  23,
    "timestamp":  238778621,
}
```

Как видите, это недействительный блок; давайте попробуем загрузить его в валидатор Block():

```
Block().load(some_block)

marshmallow.exceptions.ValidationError: {'transactions':
{'_schema': ['Invalid input type.']}, 'previous_hash': ['Not a
valid string.'], 'nonce': ['Not a valid string.']}
```

Как видите, Marshmallow довольно описателен и точно говорит нам, что не так с блоком. Попробуем еще раз, исправим ошибки:

```
some_block = {
    "mined_by": "some public key",
    "transactions": [],
    "height": 123,
    "difficulty": 10,
    "hash": "some fake hash",
    "previous_hash":  0,
    "nonce":  23,
    "timestamp":  238778621,
}
Block().load(some_block)

ValidationError: {'_schema': ['Fraudulent block:
hash iswrong']}
```

Как видите, Marshmallow запустил нашу функцию проверки, которая правильно определила, что наш хэш «некоего фальшивого хэша» явно неверен. Давай попытаемся снова:

```
some_block = {
    "mined_by": "some public key",
    "height": 123,
    "difficulty": 10,
    "previous_hash": "some previous hash",
    "nonce": "213",
    "timestamp": 238778621,
    "hash": "a52bfa60bc4ad3d3e9571eab8b28370166f2476e0f1026df
            219bec07a0a9e2e7"
}

# Passes validation by not throwing an exception
Block().load(some_block)
```

Теперь мы можем использовать вспомогательный метод для создания сообщения:

```
from funcoin.messages import create_block_message

some_block = {
    "mined_by": "some public key",
    "height":  123,
    "difficulty": 10,
    "previous_hash": "some previous hash",
    "nonce":  "213",
    "timestamp":      238778621,
    "hash": "a52bfa60bc4ad3d3e9571eab8b28370166f2476e0f1026df
    219bec07a0a9e2e7"
    }
    message = create_block_message(some_block, "127.0.0.1", 8888)
```

Теперь мы можем отправить сообщение нашим пирам и, таким образом, завершить конкретизацию модуля funcoin/peers. py module.

Объединяя все это

Прежде чем мы сможем запустить наш узел, нам нужно сделать еще немного ручной работы:

1. Нам нужно найти способ определить наш внешний IP-адрес, это IP-адрес, который виден внешнему миру.
2. Найдите способ «инициализировать» одноранговые узлы на нашему узле, когда он загружается: в конце концов, нам нужно заполнить сеть.
3. Решите, будет ли наш узел майнером; если да, то нам нужно инициировать начало добычи (а также распространение блоков среди наших пиров, когда они будут найдены).

Давайте последовательно рассмотрим указанные пункты.

Как узнать свой внешний IP-адрес

Есть сторонние сервисы, которые при подключении к ним сообщают ваш внешний IP-адрес. Это может показаться странным, но именно так биткойн-узлы находят свои собственные внешние IP-адреса.

Откройте ваше окно терминала и воспользуйтесь cURL для доступа к сервису ipinfo.io:

```
curl ipinfo.io
{
  "ip": "72.81.18.117",
  "hostname": "XX-XX-XXX-XXX.com",
  "city": "New York City",
  "region": "New York",
  "country": "US",
  "loc": "41.5143,-73.8060",
```

```
  "org": "AS701 MCI Communications Services, Inc. d/b/a Verizon
         Business",
  "postal": "10004",
  "timezone": "America/New_York",
  "readme": "https://ipinfo.io/missingauth"
}
```

Обратите внимание, что он показывает мне, какой у меня внешний IP-адрес (72.81.18.117). Давайте напишем для этого метод на Python, когда наш узел загружается в funcoin/utils.py:

```
import aiohttp
import structlog

logger = structlog.getLogger(__name__)

async def get_external_ip():
    async with aiohttp.ClientSession() as session:
        async with session.get('http://ipinfo.io',
        headers={"user-agent": "curl/7.64.1"}) as response:
            response_json = await response.json(content_
            type=None)
            ip = response_json["ip"]
            logger.info(f"Found external IP: {ip}")
            return ip
```

Когда мы запускаем метод `listen()` нашего сервера в server.py, мы можем заполнить IP-адрес.

Листинг 7-9. funcoin/server.py

```
1  async def listen(self, hostname="0.0.0.0", port=8888):
2      server = await asyncio.start_server(self.handle_
       connection, hostname, port)
3      logger.info(f"Server listening on {hostname}:{port}")
4
5      self.external_ip = await self.get_external_ip()
```

```
6       self.external_port = 8888
7
8       async with server:
9           await server.serve_forever()
```

Давайте попробуем загрузить наш сервер первый раз:

```
(venv) $ python node.py
2020-05-18  02:42.28 Creating genesis block
2020-05-18 02:42.28 Server listening on 0.0.0.0:8888
2020-05-18 02:42.28 Found external IP: 72.81.18.117
```

Вау, как это здорово? Наш сервер теперь работает в сети. Но пиры не смогут подключиться к нам по нескольким причинам:

1. У вас включен брандмауэр (и если вы действительно не знаете, что делаете, вы должны оставить его включенным).
2. Ваш маршрутизатор/точка доступа Wi-Fi не перенаправляет внешние соединения на ваш локальный компьютер (на котором вы, вероятно, его запустили), и он не должен этого делать (это серьезная уязвимость в системе безопасности).
3. Одноранговые узлы в сети не могут найти нас, потому что пока мы являемся единственным узлом.

Примечание Предыдущие причины должны отобразить четкую картину того, насколько сложно на самом деле построить одноранговую сеть. Помимо проблем с безопасностью, связанных с открытием входящего интернет-трафика, также очень сложно создать сеть постоянно подключенных к сети пиров. Если вы серьезно относитесь к тестированию, я предлагаю не пожалеть нескольких долларов в месяц за дроплеты AW eC2 или digitalocean (рекомендую) и запускать узел в контексте, в котором риск атаки невелик.

ГЛАВА 8

Сравнение с реальными децентрализованными сетями

Поздравляем, у вас есть рабочий узел (и вы фанкойнер). Но насколько это все отличается от Биткойна? Или Ethereum? Или Monero? Или какой-то еще? Какие есть альтернативы PoW? Что такое смарт-контракты? В этой главе мы рассмотрим различия между фанкойнами и реальными коммерческими блокчейнами, а также попытаемся количественно определить дистанцию между ними.

Почему разработка блокчейна сложна

Блокчейн-инженерия порождает одни из самых сложных проблем в области информатики, главным образом потому, что блокчейны децентрализованы: они связывают воедино концепции распределенных систем, сетей, параллелизма, криптографии, экономики и теории игр. И любая работа по исправлению ошибок или недостатков должна выполняться в контексте работающей

системы без ущерба для удобства ее использования. Более того, любой клиент с новым программным обеспечением должен по-прежнему поддерживать возможность проверки блоков, созданных старым программным обеспечением, создавая постоянно расширяющийся список операторов if — это сродни устранению проблем на большом самолете, когда самолет с пассаирами находится в полете.

В основе программного обеспечения с открытым исходным кодом лежит идея о том, что исправления и изменения вносятся любым заинтересованным лицом. Биткойн интересен с точки зрения теории игр, потому что со временем, по мере того, как в него будет вкладываться все больше стоимости, недреманный взгляд инвесторов должен быть в состоянии как моно быстрее реагировать на угрозы и исправления. Но, конечно, не так много сществует приложений, которые придают такое повышенное значение обратной совместимости, как Биткойн. Поскольку одна единственная ошибка может привести к тому, что все это богатство (на момент написания этой книги — 120 миллиардов долларов) исчезнет, философия Биткойна состоит в том, чтобы действовать осторожно и достичь консенсуса сообщества разработчиков, прежде чем удваивать ставки. В этом лане хорошая мера еще и еще раз перепроверять на наличие ошибок. А в некоторых случаях ошибки просто не стоит исправлять, поскольку стоимость исправления просто не стоит хлопот с обновлением. Например, в биткойне Сатоши сам представил известную ошибку искажения времени, из-за которой блоки всегда добываются чуть меньше 10 минут (вместо 10 минут): вместо усреднения времени, затраченного на добычу последних 2016 блоков, алгоритм был выключен и смотрит на последние блоки 2015 года. Это означает, что блоки немного сложнее добывать, чем это должно быть. Это не считается серьезной проблемой, но стоимость ее устранения значительно перевешивает преимущества.

В общедоступных блокчейнах изменения рдоставляются одним

из двух сособов: хардфорками и софтфорками. Хардфорки — это отклонения от текущей цепочки блоков. Такое отклонение приводит к тому, что старые клиенты больше не могут понимать новый протокол, за которым следуют новые клиенты. Bitcoin Cash, Bitcoin Gold, Bitcoin XT и Bitcoin Classic — все это примеры хардфорков Биткойна. С другой стороны, софтфорки обратно совместимы: новые блоки, сгенерированные новыми клиентами, могут быть проверены старыми клиентами.

Не путайте хардфорк (изменение правил консенсуса, которое нарушает безопасность узлов, которые не обновляются), софтфорк (изменение правил консенсуса, которое ослабляет безопасность узлов, которые не обновляются), программный форк (когда один или несколько разработчиков постоянно разрабатывают кодовую базу отдельно от других разработчиков) и гитфорк (когда один или несколько разработчиков временно разрабатывают кодовую базу отдельно от других разработчиков).

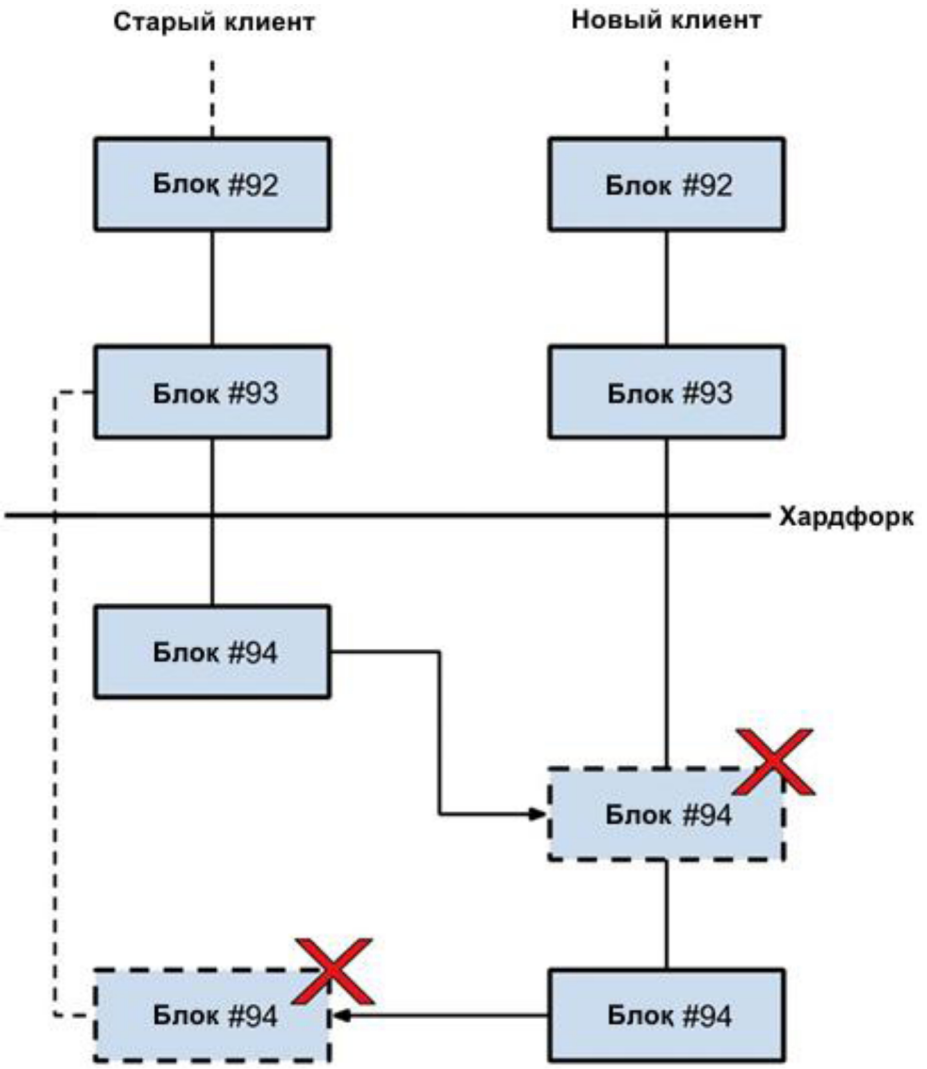


Рис. 8-1. *Хардфорк*

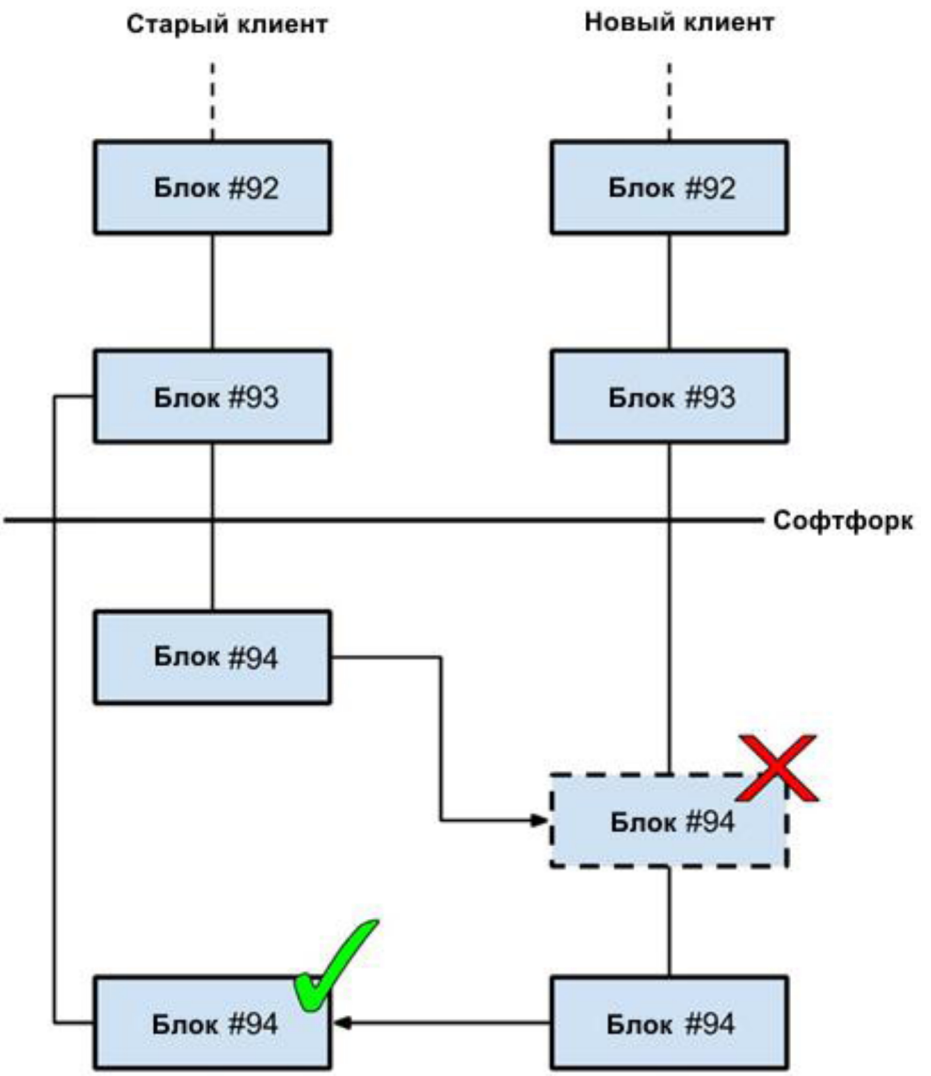


Рис. 8-2. *Софтфорк*

Примчание Другой тип форка — это программный форк, поскольку кодовые базы имеют открытый исходный код, ничто не мешает вам «разветвить» кодовую базу Биткойна на свою собственную. Существует множество проектов, которые регулярно этим занимаются, наиболее заметным из которых является лайткоин.

Недостатки фанкоина

Как вы убедились на собственном опыте, существует примерно четыре критических компонента, необходимых для того, чтобы иметь *публично* функционирующую цепочку блоков:

- **Протокол консенсуса**, который позволяет распределенным одноранговым узлам согласовывать состояние системы (данные, хранящиеся в цепочке блоков). Именно эта магия позволяет нам называть блокчейн «децентрализованным».
- **Сетевой уровень**, который позволяет одноранговым узлам общаться и распространять информацию по сети.
- **Цепочка криптографически защищенных блоков** (сама цепочка блоков).
- **Экономичная схема поощрения** (доказательство работы), которая защищает цепочку под давлением работы.

Проблемы, возникающие в любом из этих компонентов, почти всегда имеют последствия для *всех* из них. Например, ошибка на сетевом уровне может быть использована для задержки распространения транзакций на определенные узлы в сети, тем самым влияя на протокол консенсуса. Написание этой книги было для меня чрезвычайно сложным, потому что я намеревался

построить простую для понимания цепочку блоков, которая была бы проста для понимания, только для того, чтобы постепенно осознать, что такие проекты, как Биткойн, имеют ту же цель: в некотором смысле они являются простейшим проявлением их *протоколов*.

В качестве мысленного упражнения я бы хотел, чтобы вы на мгновение задумались о работе, проделанной в различных компонентах funcoin, и подумали о том, насколько они устойчивы к атакам. Сетевой уровень — сложная и важная грань нашей системы. С него, наверное, и начнем.

Сетевой уровень

Первым предположением, которое мы сделали на сетевом уровне, было использование JSON в качестве формата сериализации наших сообщений. Это неоптимально по множеству причин, в основном из-за того, что не поддерживает потоковую передачу. Другими словами, нам нужно знать, где начинается один объект в кодировке JSON и начинается другой. Чтобы обойти это препятствие, мы установили правило, согласно которому сообщения разделяются символами новой строки. Вот код в нашем классе Server, который выполняет это:

```
1  async def handle_connection(self, reader:
       StreamReader, writer: StreamWriter):
2      while True:
3          try:
4              # Ждем, пока не поступят новые данные
5              data = await reader.readuntil(b"\n")
6
7              decoded_data = data.decode("utf8").strip()
8
9              try:
10                 message = BaseSchema().loads(decoded_data)
```

```
11          except  MarshmallowError:
12                  logger.info("Received unreadable message",
                    peer=writer)
13                  break
14
15              # Извлекаем адрес из сообщения, добавляем его в
                  объект записи
16              writer.address = message["meta"]["address"]
17
18              # Давайте добавим пир в наш пул соединений
19              self.connection_pool.add_peer(writer)
20
21              # ... и обработаем сообщение
22              await self.p2p_protocol.handle_message
                (message, writer)
23
24              await writer.drain()
25              if writer.is_closing():
26                  break
27
28          except (asyncio.exceptions.IncompleteReadError,
            ConnectionError):
29              # Произошла ошибка, выход из цикла ожидания
30              break
```

Вы видите в этом проблему? Это оставляет наш узел открытым для атаки типа «отказ в обслуживании». Все, что кому-то нужно сделать, это отправить нам сообщение без новой строки, и наш узел будет держать соединение открытым, ожидая окончания сообщения. Это, однако, можно легко устранить, отслеживая и ограничивая объем данных, которые может отправить нам одноранговый узел. При этом мы использовали сообщения на основе JSON, чтобы упростить реализацию. Биткойн использует байты с разделителями столбцов для структур сообщений, что

требует большой трудоемкости реализации. Вот структура биткойн-сообщения:

Размер поля	Описание	Тип данных	Коммнтарий
4	magic	uint32	означает, должно ли сообщение быть в тестовой или реальной сети
12	command	char[12]	строка, обозначающая тип содержимого сообщения
4	length	uint32	размер полезной информации в байтах
4	checksum	uint32	первые 4 байта хэша сообщения
?	payload	char[]	фактические данные

Мы также не говорили о том, как наш узел сортирует и хранит пиры. У нас нет ***эвристики*** для этого. Позвольте мне привести пример того, что я имею в виду под эвристикой: когда одноранговый узел запрашивает у нас узлы, мы могли бы отправить ему ***все*** узлы, о которых мы знаем, но это расточительно и может быть оптимизировано. Что, если мы отправим запрашивающему узлу 20 пиров, выбранных случайным образом?

Приведет ли это к тому, что одноранговый узел сможет распространить транзакцию на всю сеть (при условии, что другие одноранговые узлы также возвращают случайные узлы)? В основном это относится к сфере одноранговых исследований, и такие эвристики обычно называют *протоколами Gossip* или *одноранговым обнаружением*. Интерес к этой области исследований возобновился после знаменитого закрытия Napster — первого программного обеспечения, позволявшего людям загружать музыку из одноранговой сети. Но Napster был централизован: он обслуживал список доступных пиров, поэтому, если Napster отключался, то же самое происходило и с

сетью.

BitTorrent является примером распределенной хэш-таблицы (DHT): децентрализованной системы хранения значений ключей, обычно используемой для нелегальной загрузки фильмов и телепередач через такие веб-сайты, как The Pirate Bay и ISOHunt.

Правительства и правоохранительные органы в течение многих лет пытались закрыть эти так называемые торрент-сайты, но в основном безуспешно — они существуют до сих пор. Но почему? Что делает их такими устойчивыми к цензуре? Похоже, блокчейны, такие как Биткойн и Ethereum, должны кое-чему научиться у DHT, таких как eDonkey и Kad. Секрет заключается в том, как узлы в этих сетях обнаруживают и организуют свои одноранговые узлы — правильное выполнение этого делает одноранговую сеть масштабируемой, устойчивой к цензуре и быстрой. Если вашей сети необходимо поддерживать хранилище файлов, вы можете рассмотреть возможность использования DHT, поскольку на самом деле это просто правила для управления тем, какие узлы в сети размещают какие файлы. Если вы строите децентрализованную цепочку блоков, вам не нужно беспокоиться о хранении больших файлов, поскольку все действительно хранят один и тот же «большой» файл — саму цепочку блоков.

Большая часть исследований одноранговых протоколов находится в зачаточном состоянии и связана с обменом файлами. Современные децентрализованные сети существуют на уровне приложений и реализуют протоколы для

- **Одноранговое обнаружение**: Как обнаружить узлы в сети
- **Маршрутизация**: Какие маршруты выбрать для контактных узлов (сообщений)
- **Присвоение имен**: Единый способ идентификации узлов

Эти протоколы почти всегда существуют на прикладном уровне сетевого стека.

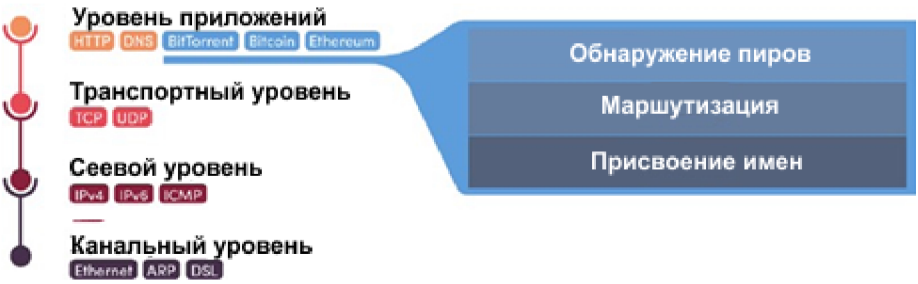


Рис. 8-3. Типичный сетевой стек

Одноранговое обнаружение

Современные одноранговые сети не централизованы: Было время, когда они были такими — когда BitTorrent использовал трекеры (серверы, имевшие множество IP-адресов для подключения, например, Napster). Но если бы одноранговые сети полагались на трекеры, они были бы подвержены простой форме атаки: отключению трекера.

Представьте, что мы с вами хотим общаться друг с другом через Интернет без сервера. Как мы находим IP друг друга?

> Давайте использовать электронную почту/SMS/Telegram/WhatsApp для отправки наших IP-адресов друг другу?

Это могло бы сработать, но этот способ **централизованный** — если электронная почта не работает, система не работает (интересно, что в первые дни Биткойн использовал IRC-чаты для перечисления IP-адресов).

> Мы можем договориться запускать наши клиенты по определенному порту и продолжать пинговать случайные IP-адреса по этому порту, пока не найдем друг друга.

Найти друг друга маловероятно — нам пришлось бы пропинговать примерно 2^{32}-1 для IPV4 и 2^{128}-1 для IPv6 адресов. **Но**

мы на верном пути!

Итак, как это делает Биткойн?

Оказывается, если сеть достаточно велика, мы можем использовать комбинацию из представленных ранее способов: кодированные записи DNS. Эти записи DNS поддерживаются сообществом Биткойн и возвращают списки доверенных адресов, к которым клиент может подключиться.

Но DNS-серверы централизованы: Один сервер можно подделать и заставить возвращать поддельные IP-адреса, изолируя узел для приема поддельных транзакций и поддельной цепочки блоков. Только по этой причине клиенты не должны полагаться исключительно на DNS. Таким образом, при первом запуске клиента существует неотъемлемый риск, но он смягчается тем фактом, что чем дольше работает клиент, тем дороже обходится поддержка атаки.

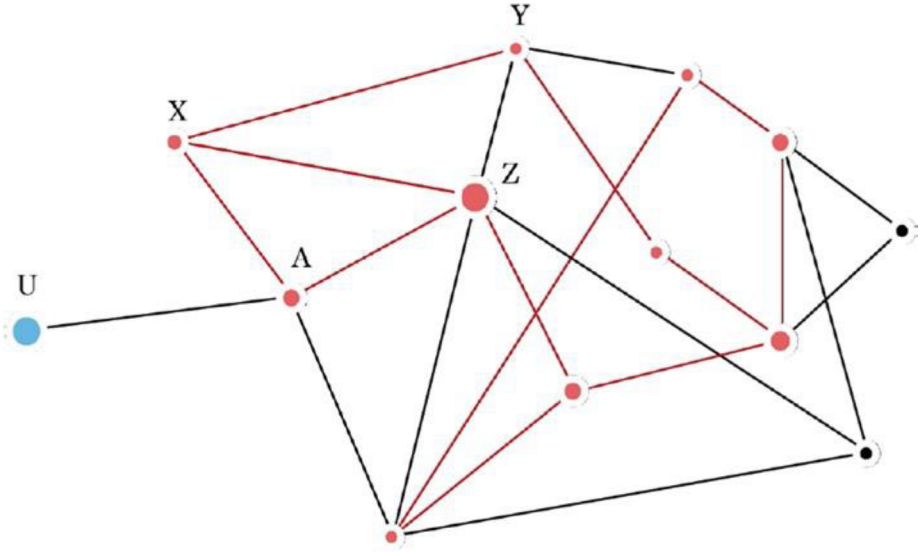


Рис. 8-4. *Узел, впервые подключающийся к сети*

На предыдущей диаграмме, когда наш биткойн-клиентский узел U находит другой узел, он начинает с ним общаться:

Здравствуйте, узел A, я использую протокол Биткойн v0.16.0

затем узел A отвечает:

Эй узел U! Я также использую v0.16.0, мой последний блок - 528491, и вот некоторые другие узлы, о которых я знаю: узел X, узел Z,...

Затем узел A передает нас другим узлам, которые также могут связаться с нами. И мы можем загрузить недостающие блоки из нашего нового *роя* узлов и запросить у них информацию об их соседях.

На этом основано обнаружение одноранговых узлов и устойчивость одноранговых стей — здесь нет центральной точки отказа, есть только «облако» узлов. А остановка хорошо организованной сети сродни цензуре всех сетевых подключений по-отдельности — невыполнимая задача.

Присвоение имен

IP-адрес пира может меняться, поэтому нам нужен способ постоянно ссылаться на него в сети. Ethereum использует 256-битный хэш открытого ключа пира для идентификации пира. BitTorrent использует 160-битный хэш случайного числа. В любом случае идентификаторы достаточно велики, чтобы гарантировать уникальность. Если в сети используется соответствующая хеш-функция, скажем, SHA-256, то мы можем разместить $\sim 2^{256}$ одноранговых узлов в нашей сети.

Функции дистанции

Прежде чем мы сможем организовать нашу сеть, нам нужен способ измерения «дистанции» между двумя одноранговыми узлами в сети. В математике это называется метрикой. Что означает «способ измерения расстояния». Важно понимать, что мы не имеем в виду географическое (евклидово) расстояние, мы говорим о чем-то

абстрактном, общем способе сети определить, насколько близко два узла находятся друг от друга.

В 2002 году два исследователя Петар Маймунков и Давид Мазьер опубликовали новаторскую статью под названием Kademlia. В нем описывается, как при предоставлении определенного способа измерения расстояния децентрализованная сеть может быть структурирована таким образом, чтобы она стала быстрой, отказоустойчивой и маршрутизируемой. Большинство DHT, таких как BitTorrent, используют алгоритм Kademlia.

Давайте немного поговорим о том, как это работает.

Алгоритм Kademlia использует функцию XOR (исключающее или) для измерения расстояния между любыми двумя узлами. В качестве повторения, XOR — это логическая операция между двумя данными на входе логического оператора, которая выводит состояни *Истина* только в том случае, если данные на входе не одинаковые. Вот несколько примеров:

```
1000 XOR 1000 = 0000
0000 XOR 1111 = 1111
1011 XOR 0101 = 1110
```

В математике, чтобы называться метрикой, функция дистанции должна удовлетворять следующим трем критериям (как и метрика XOR):

- Расстояние узла между самим собой равно нулю.
- Расстояние между узлом A и узлом B такое же, как и расстояние между узлом B и узлом A (симметрия).
- Расстояние от узла A до узла B и узла C всегда больше или равно расстоянию от узла A до узла C (неравенство треугольника).

Кроме того, функция XOR проста в реализации и дешева в вычислениях. Важно понимать, что функция XOR просто

вычисляет расстояние между двумя идентификаторами узлов.

Маршрутизация

На следующем изображении показан граф небольшой сети с максимум восемью узлами. Идентификаторы узлов перечислены в виде листьев в нижней части диаграммы. На изображении показано, как устроена сеть с точки зрения узла 110. Как вы можете видеть, ближайший узел — 111, поскольку 110 XOR 111 = 001. Самый дальний узел — 001, поскольку 110 XOR 001 = 111. Как видите, есть несколько замечательных свойств использования XOR в качестве функции расстояния; во-первых, это гарантирует, что половина сети хранится в самой дальней группе (или в корзине, как это упоминается в статье). Это удобно, потому что если одноранговый узел запрашивает у узла 110 своих соседей, узлу 110 нужно вернуть запрашивающему узлу только ближайшие одноранговые узлы.

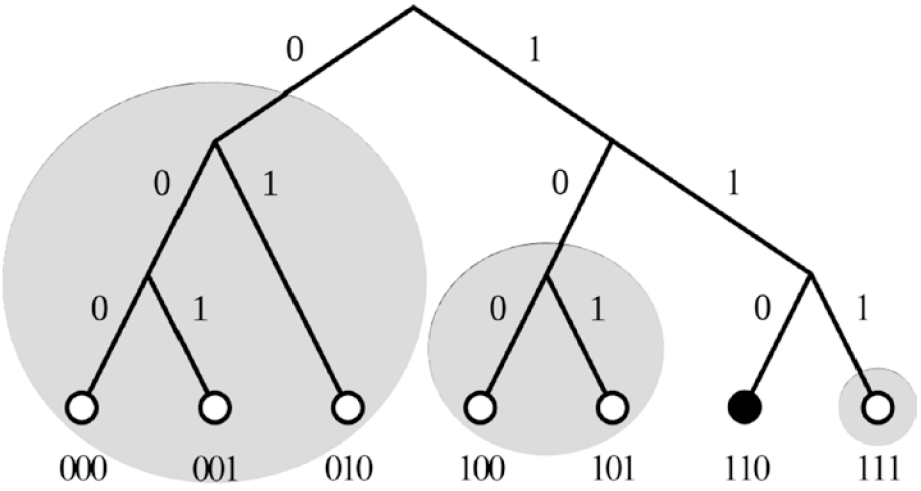


Рис. 8-5. *Алгоритм Kademlia*

Постоянство данных

Наш блокчейн хранится в памяти, то есть наш класс Blockchain очищает и поддерживает всю цепочку в работающей программе!

```
class  Blockchain(object):
    def_init_(self):
        self.chain = []
```

Это означает, что каждый раз, когда мы перезагружаем сервер, нашему узлу приходится загружать весь блокчейн от своих соседей. Блокчейн монотонен — он постоянно увеличивается в размерах, поэтому нам нужно найти лучшее решение для его хранения. Биткойн использует базу данных Google LevelDB для локального хранения. LevelDB — это простое хранилище значений ключей, которое можно сохранить на диске. На момент написания книги полный объем хранилища Биткойн составляет около 160 Гб.

Альтернативный консенсус: Доказательство участия

В этой книге мы говорили только о доказательстве работы: алгоритме, который защищает нашу цепочку блоков и создает консенсус. Стоит отметить, что существует громкая критика в адрес доказательства работы, наиболее популярным из которых является аргумент о том, что это вредно для окружающей среды: не секрет, что Биткойн потребляет огромное количество энергии для доказательства работы. Но это в значительной степени оспаривается аргументом теории игр, согласно которому именно эта энергия защищает сеть. Кроме того, он стимулирует майнеров биткойнов оставаться прибыльными, перемещая свои операции по добыче в штаты, которые имеют излишки электроэнергии или недорогие возобновляемые источники

энергии. На макроуровне биткойн-максималисты противопоставляют потребление энергии неопределимым затратам на управление фиатной банковской системой с бесчисленными филиалами, хранилищами, электричеством, сотрудниками, формами и, казалось бы, бесконечным запасом бумажных банкнот.

В доказательстве участия или Proof of Stake (PoS) майнер следующего блока выбирается алгоритмом в соответствии с определенными критериями. В зависимости от реализации эти критерии могут быть связаны с состоянием майнера или периодом времени, в течение которого он поддерживал баланс, или могут быть совершенно случайными. Кроме того, PoS не требует чрезмерного использования энергии для защиты сети. NXT — это криптовалюта, в которой реализовано чистое Proof of Stake; другие криптовалюты, такие как Peercoin, используют гибридную систему, в то время как (на момент написания книги) Ethereum находился в процессе перехода к системе PoS.

В сообществе есть множество разработчиков, которые не считают Proof of Stake идеальной заменой Proof-of-Work из-за множества факторов. Во-первых, доказательство работы предлагает дополнительные преимущества на сетевом уровне: если Биткойн подвергнется атаке затмения — тысячи узлов злоумышленников в сети, пытающихся создать альтернативную цепочку, — то эти агрессоры будут вынуждены повторить всю работу, проделанную с помощью триллионы тераватт-часов электроэнергии, что делает атак невозможной; во-вторых Proof of Stake подвержен проблеме, известной как проблема «ничего на кону», когда майнерам (возможно, лучший термин — генераторам блоков) нечего терять, действуя недобросовестно (голосуя за несколько историй блокчейна) и недопуская консенсус. Это прямое следствие отсутствия необратимых экономических затрат (например, затрат энергии), поэтому работа над несколькими цепями одновременно не требует затрат. Чтобы уменьшить эти уязвимости, были предложены карательные протоколы, которые наказывают злоумышленников в сети.

Смарт-контракты

Идея «умного» контракта была первоначально представлена биткойнскими P2PK (Pay to Pubkey) и P2PH (Pay to Pubkey Hash). Прорывом Ethereum стало введение совместимого по Тьюрингу языка для написания смарт-контрактов. Проще говоря, совместимость по Тьюрингу означает, что язык смарт-контрактов может эффективно выполнять операции, как любой реальный компьютер.

Внедрение смарт-контрактов изначально рассматривалось как более полная версия системы скриптов Биткойн или, проще говоря, кода, который может храниться в блокчейне и выполняться майнерами. Это называется ***контрактом***, потому что идея кода заключается в том, что он работает с учетными записями или служит посредником, регистрируя события в блокчейне или предпринимая действия при соблюдении определенных критериев. Это особенно уместно в правовом пространстве, поскольку уменьшает потребность в редставитлях или посредниках. В funcoin наш блокчейн хранит простые транзакции, а не инструкции (или код). Технически смарт-контракт вовсе не так уж умен — это набор кода, написанного на специальном языке, который майнеры (и узлы) запускают при проверке блока. Этот код может указывать на перевод средств с одного счета на другой или разрешать выполнение определенной транзакции по запросу; возможности кажутся безграничными.

Пример 1. Нотариальное заверение документа

Допустим, вы хотите нотариально заверить документ. Сначала вам нужно поручить своему банку заплатить вашему адвокату, а затем подождать, пока ваш адвокат обработает ваш документ. Однако со смарт-контрактом нет необходимости в банках или каком-либо посредничестве. Благодаря смарт-контрактам юрист может получать оплату автоматически после того, как он

создаст документ, удовлетворяющий определенным условиям, установленным вашим кодом.

Пример 2. Покупка дома

Покупка недвижимости – очень сложная процедура. Обычно существуют комиссионные брокеры, представляющие продавца и покупателя. Они координируют свои действия друг с другом, а также со своими юристами и банками для управления процессом налогообложения. В некоторых случаях деньги также должны быть помещены на счет условного депонирования, чтобы перевести средства от покупателя к продавцу, когда все документы будут подписаны. Этот процесс завоевал огромное доверие, он облегчает то, что по сути является простой транзакцией. В этом случае смарт-контракт может автоматически выступать как в качестве услуги условного депонирования, так и в качестве провайдера платежей, поскольку все стороны могут автоматически заключать сделки при соблюдении критериев.

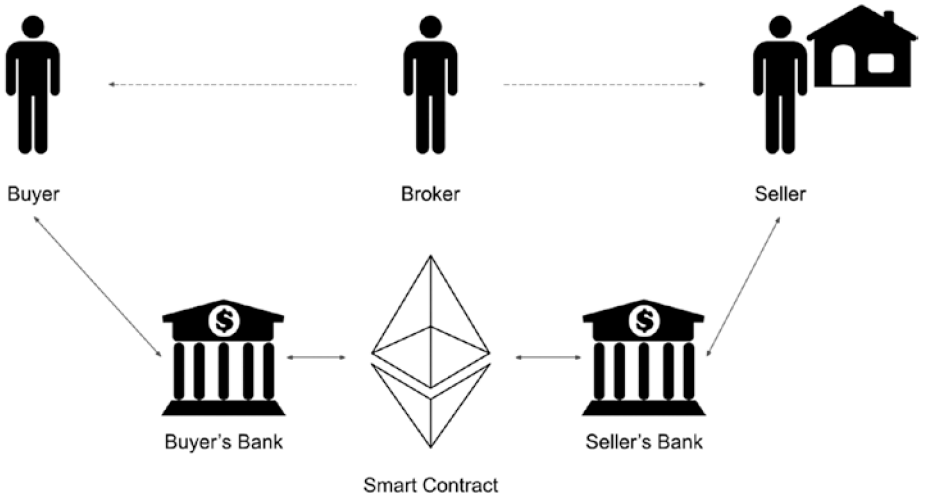


Рис. 8-6. *Пример смарт-контракта*

Пример 3. ICO (первоначальные предложения монет)

ICO можно рассматривать как краудфандинг на основе блокчейна. Если вы хотите продать доли в своем новом проекте или компании, вы можете создать смарт-контракт, который предоставляет кому-то долю при переводе монет на конкретный счет. Эти акции обычно представлены в виде токенов.

Как выглядит смарт-контракт?

В Ethereum смарт-контракты написаны на языке под названием Solidity. Вот пример контракта на голосование (из документации Solidity на сайте https://github.com/ethereum/solidity/blob/v0.4.20/docs/solidity-by-example.rst):

```
/// @title Voting with delegation.
    contract Ballot {
        // This declares a new complex type which will
        // be used for variables later.
        // It will represent a single voter.
        struct Voter {
            uint weight; // weight is accumulated by delegation
            bool voted;  // if true, that person already voted
            address delegate; // person delegated to
            uint vote; // index of the voted proposal
        }

        // This is a type for a single proposal.
        struct Proposal {
            bytes32 name;   // short name (up to 32 bytes)
            uint voteCount; // number of accumulated votes
```

```
    }

    address public chairperson;

    // This declares a state variable that
    // stores a `Voter` struct for each possible address.
    mapping(address => Voter) public voters;

    // A dynamically sized array of `Proposal` structs.
    Proposal[] public proposals;

    /// Create a new ballot to choose one of `proposalNames`.
    constructor(bytes32[] memory proposalNames) public {
        chairperson = msg.sender;
        voters[chairperson].weight = 1;

        // For each of the provided proposal names,
        // create a new proposal object and add it
        // to the end of the array.

     for (uint i = 0; i < proposalNames.length; i++) {
         // `Proposal({...})` creates a temporary
         // Proposal object and `proposals.push(...)`
         // appends it to the end of `proposals`.
         proposals.push(Proposal({
             name:  proposalNames[i],
             voteCount: 0
         }));
     }
}

// Give `voter` the right to vote on this ballot.
// May only be called by `chairperson`.
function giveRightToVote(address voter) public {
    // If the first argument of `require` evaluates
    // to `false`, execution terminates, and all
```

```
        // changes to the state and to Ether balances
        // are reverted.
        // This used to consume all gas in old EVM versions but
        // not anymore.
        // It is often a good idea to use `require` to check if
        // functions are called correctly.
        // As a second argument, you can also provide an
        // explanation about what went wrong.
        require(
            msg.sender == chairperson,
            "Only chairperson can give right to vote."
        );
        require(
            !voters[voter].voted,
            "The voter already voted."
        );
        require(voters[voter].weight == 0);
        voters[voter].weight = 1;
    }

    /// Delegate your vote to the voter `to`.
    function delegate(address to) public {
        // assigns reference
        Voter storage sender = voters[msg.sender];
        require(!sender.voted, "You already voted.");

        require(to != msg.sender, "Self-delegation is
        disallowed.");

        // Forward the delegation as long as
        // `to` is also delegated.
        // In general, such loops are very dangerous,
        // because if they run too long, they might
        // need more gas than is available in a block.
```

```
        // In this case, the delegation will not be executed,
        // but in other situations, such loops might
        // cause a contract to get "stuck" completely.
        while (voters[to].delegate != address(0)) {
            to = voters[to].delegate;

            // We found a loop in the delegation, not allowed.
            require(to != msg.sender, "Found loop in
            delegation.");
        }

        // Since `sender` is a reference, this
        // modifies `voters[msg.sender].voted`
        sender.voted = true;
        sender.delegate = to;
        Voter storage delegate_ = voters[to];
    if (delegate_.voted) {
        // If the delegate already voted,
        // directly add to the number of votes
        proposals[delegate_.vote].voteCount += sender.
        weight;
    } else {
        // If the delegate did not vote yet,
        // add to her weight.
        delegate_.weight += sender.weight;
    }
}

/// Give your vote (including votes delegated to you)
/// to proposal `proposals[proposal].name`.
function vote(uint proposal) public {
    Voter storage sender = voters[msg.sender];
    require(sender.weight != 0, "Has no right to vote");
    require(!sender.voted, "Already voted.");
    sender.voted = true;
```

```
        sender.vote = proposal;

        // If `proposal` is out of the range of the array,
        // this will throw automatically and revert all
        // changes.
        proposals[proposal].voteCount += sender.weight;
    }

    /// @dev Computes the winning proposal taking all
    /// previous votes into account.
    function winningProposal() public view
            returns (uint winningProposal_)
    {
        uint winningVoteCount = 0;
        for (uint p = 0; p < proposals.length; p++) {
            if (proposals[p].voteCount > winningVoteCount) {
                winningVoteCount = proposals[p].voteCount;
                winningProposal_ = p;
            }
        }
    }

    // Calls winningProposal() function to get the index
    // of the winner contained in the proposals array and then
    // returns the name of the winner
    function winnerName() public view
            returns (bytes32 winnerName_)
    {
        winnerName_ = proposals[winningProposal()].name;
    }
}
```

Когда этот код выполняется майнером, вычисление измеряется в *газе*. Другими словами, *газ* — это расчет количества вычислительных усилий для выполнения контракта. Каждая операция требует

определенного количества газа, и майнеры получают комиссию (в Ethereum) за общее количество потраченного газа. Эта экономия стимулирует эффективное составление контрактов.

ПРИЛОЖЕНИЕ А

Биткойн: одноранговая электронная кассовая система Сатоши Накамото

Это оригинальный технический документ, полностью воспроизведенный в том виде, в каком он был опубликован Сатоши Накамото 31 октября 2008 г.

Резюме

Чисто одноранговая версия электронных денег позволит отправлять онлайн-платежи напрямую от одной стороны к другой, минуя финансовое учреждение. Цифровые подписи являются частью решения, но основные преимущества теряются, если для предотвращения двойных оплат по-прежнему требуется доверенная третья сторона. Мы предлагаем решение проблемы двойной оплаты с помощью одноранговой сети. Сеть присваивает

транзакциям временные метки, хэшируя их в непрерывную цепочку подтверждения работы на основе хэша, формируя запись, которую нельзя изменить без повторного выполнения проверки работы. Самая длинная цепочка служит не только доказательством последовательности наблюдаемых событий, но и доказательством того, что она возникла из-за наибольшего пула мощности ЦП. Пока большая часть мощности ЦП контролируется узлами, которые не сотрудничают для организации атаки на сеть, они будут генерировать самую длинную цепочку и опережать злоумышленников. Сама сеть требует минимальной структуры. Сообщения передаются в режиме максимальных усилий, и узлы могут покидать сеть и присоединяться к ней по своему желанию, принимая самую длинную цепочку проверки работоспособности в качестве доказательства того, что произошло, пока они отсутствовали.

Введение

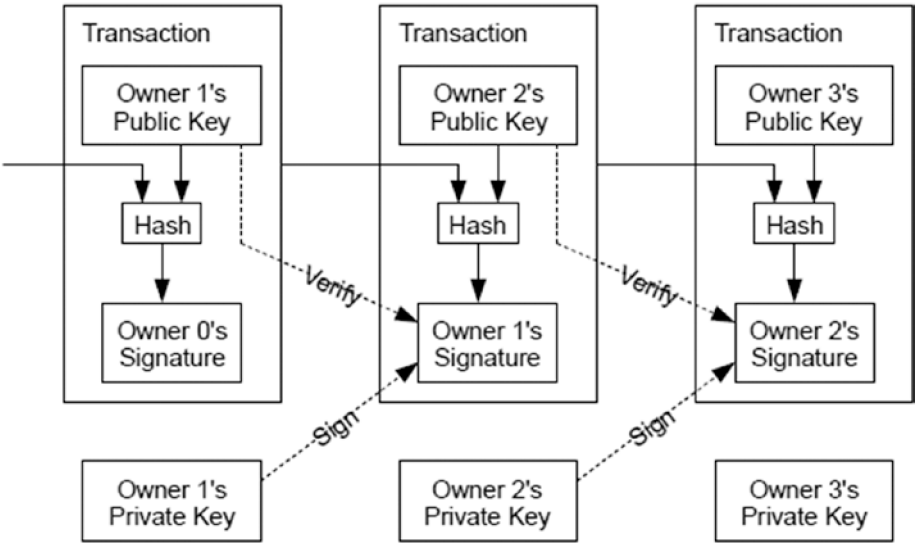
Торговля в Интернете стала полагаться почти исключительно на финансовые учреждения, выступающие в качестве доверенных третьих сторон для обработки электронных платежей. Хотя система работает достаточно хорошо для большинства транзакций, она по-прежнему страдает недостатками модели, основанной на доверии. Полностью необратимые транзакции на самом деле невозможны, поскольку финансовые учреждения не могут избежать посредничества в спорах. Стоимость посредничества увеличивает транзакционные издержки, ограничивая минимальный практический размер транзакции и отсекая возможность для небольших случайных транзакций, а потеря возможности осуществлять необратимые платежи за необратимые услуги сопряжена с более широкими издержками. При возможности обратимости потребность в доверии расширяется. Продавцы

должны с осторожностью относиться к своим покупателям, выпрашивая у них больше информации, чем им в противном случае потребовалось бы. Определенный процент мошенничества принимается как неизбежный. Этих затрат и неопределенности платежей можно избежать лично, используя физическую валюту, но не существует механизма для осуществления платежей по каналу связи без доверенной стороны.

Что необходимо, так это система электронных платежей, основанная на криптографическом доказательстве, а не на доверии, позволяющая любым двум сторонам совершать сделки напрямую друг с другом без необходимости в доверенной третьей стороне. Транзакции, которые невозможно отменить с вычислительной точки зрения, защитят продавцов от мошенничества, а обычные механизмы условного депонирования могут быть легко реализованы для защиты покупателей. В этой статье мы предлагаем решение проблемы двойной оплаты с использованием однорангового распределенного сервера временных меток для создания вычислительного доказательства хронологического порядка транзакций. Система безопасна до тех пор, пока честные узлы коллективно контролируют большую вычислитльню мощность, чем любая сотрудничающая группа узлов злоумышленников.

Транзакции

Мы определяем электронную монету как цепочку цифровых подписей. Каждый владелец передает монету следующему, подписывая цифровой подписью хэш предыдущей транзакции и открытый ключ следующего владельца и добавляя их в конец монеты. Получатель платежа может проверить подписи, чтобы проверить цепочку владения.



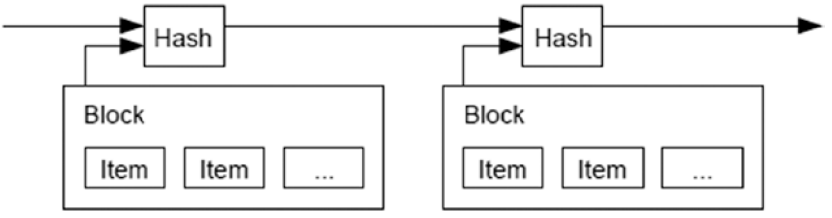
Проблема, конечно, заключается в том, что получатель платежа не может проверить, не отратил ли один из владельцев монету дважды. Распространенным решением этой проблемы является введение доверенного центрального органа, или монетного двора, который проверяет каждую транзакцию на наличие двойных платежей. После каждой транзакции монета должна быть возвращена на монетный двор для выпуска новой монеты, и только монеты, выпущенные непосредственно монетным двором, не подлежат двойному использованию. Проблема с этим решением заключается в том, что судьба всей денежной системы зависит от компании, управляющей монетным двором, и каждая транзакция должна проходить через них, как через банк.

Нам нужен такой способ, чтобы получатель платежа знал, что предыдущие владельцы не подписывали никаких более ранних транзакций. Для наших целей учитывается самая ранняя транзакция, поэтому нас не волнуют более поздние попытки двойной оплаты. Единственный способ подтвердить отсутствие транзакции — быть в курсе всех транзакций. В модели,

использующей монетный двор, последний знает обо всех транзакциях и решает, какие из них поступили первыми. Чтобы выполнить это без доверенной стороны, транзакции должны быть объявлены публично [1], и нам нужна система, позволяющая участникам согласовать единую историю порядка их получения. Получателю платежа необходимо доказательство того, что во время каждой транзакции большинство узлов согласились, что она была получена первой.

Сервер меток времени

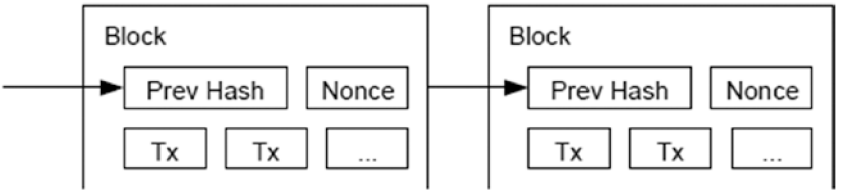
Предлагаемое нами решение начинается с сервера меток времени. Сервер меток времени работает, беря хэш блока элементов, для которого нужно поставить метку времени, и широко публикуя хеш, например, в газете или в сообщениях Usenet [2-5]. Метка времени доказывает, что для того, чтобы попасть в хэш, данные, очевидно, должны были существовать на указанный момент времени. Каждая метка времени включает предыдущую метку времени в свой хэш, образуя цепочку, где каждая дополнительная метка времени повышает надежность предыдущих.



Доказательство работы

Чтобы реализовать распределенный сервер меток времени на одноранговой основе, нам потребуется использовать систему проверки доказательства выполненной работы, аналогичную Hashcash Адама Бэка [6], а не сообщения в газетах или Usenet. Доказательство работы включает в себя сканирование значения, которое при хешировании, например, с помощью SHA-256, начинается с нулевого количества битов. Средняя требуемая работа экспоненциальна по количеству необходимых нулевых битов и может быть проверена путем выполнения одного хэша.

Для нашей сети меток времени мы реализуем доказательство работы, увеличивая одноразовый номер в блоке до тех пор, пока не будет найдено значение, которое дает хешу блока требуемые нулевые биты. После того, как ресурсы ЦП были затрачены на то, чтобы блок удовлетворял доказательству работы, он не может быть изменен без повторного выполнения работы. Поскольку более поздние блоки следуют за данным блоком, работа по изменению блока будет включать в себя повторение всех блоков после данного.



Доказательство работы также решает проблему определения представительства при принятии решений большинством. Если бы большинство основывалось на принципе «один IP-адрес — один голос», его мог бы подорвать любой, кто может выделить много IP-адресов. Доказательство работы — это, по сути, один процессор — один голос. Решение большинства представлено самой длинной цепочкой, в которую вложены наибольшие усилия по

доказательству работы. Если большая часть вычислительной мощности ЦП контролируется честными узлами, честная цепочка будет расти быстрее всех и опережать любые конкурирующие цепочки. Чтобы изменить первый блок, злоумышленнику придется переопределить доказательство работы этого блока и всех блоков цепочки после него, а затем достичь и превзойти работу честных узлов. Позже мы покажем, что вероятность того, что более медленный злоумышленник догонит их, экспоненциально уменьшается по мере добавления последующих блоков.

Чтобы компенсировать с течением времени повышение вычислительной мощности оборудования и изменение интереса к работающим узлам, сложность доказательства работы определяется скользящим средним значением среднего количества блоков в час. Если блоки генерируются слишком быстро, сложность увеличивается.

Сеть

Шаги по запуску сети следующие:

1. Новые транзакции транслируются на все узлы.
2. Каждый узел собирает новые транзакции в блок.
3. Каждый узел работает над поиском сложного доказательства работы для своего блока.
4. Когда узел находит доказательство работы, он рассылает блок всем узлам.
5. Узлы принимают блок только в том случае, если все транзакции в нем легитимны и еще не потрачены.
6. Узлы выражают свое принятие блока, работая над созданием следующего блока в цепочке, используя хэш принятого блока в качестве предыдущего хэша.

Узлы всегда считают самую длинную цепочку правильной и

будут продолжать работать над ее расширением. Если два узла одновременно транслируют разные версии следующего блока, некоторые узлы могут получить сначала один или другой блок. В этом случае узлы работают с первой полученной веткой, но сохраняют другую ветку на случай, если она станет длиннее. Связь будет разорвана, когда будет найдено следующее доказательство работы и одна ветвь станет длиннее; узлы, которые работали на другой ветке, затем переключатся на более длинную ветку.

Широковещательные рассылки новых транзакций не обязательно должны достигать всех узлов. Когда транзакции достигают большинства узлов, они вскоре попадают в блок. Блочные широковещательные рассылки также толейрантны к пропущенным сообщениям. Если узел не получил блок, он запросит его, когда получит следующий блок и определит, что пропустил один блок.

Стимул

По соглашению первая транзакция в блоке — это специальная транзакция, которая создает новую монету, принадлежащую создателю блока. Это добавляет узлам стимул поддерживать сеть и дает возможность изначально вводить монеты в обращение, поскольку нет центрального органа, который бы их выпускал. Регулярное добавление постоянного количества новых монет аналогично тому, как золотоискатели тратят ресурсы на добавление золота в обращение. В нашем случае тратится процессорное время и электроэнергия.

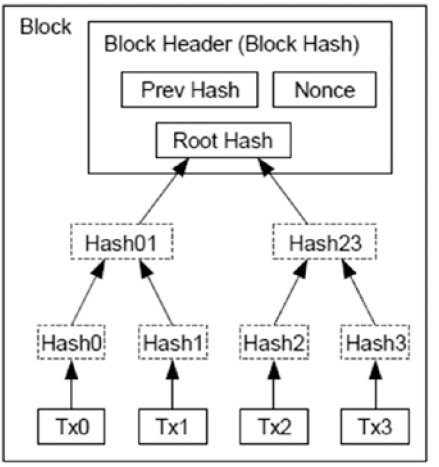
Поощрение также может быть комиссией за транзакцию. Если выходное значение транзакции меньше ее входного значения, разница представляет собой комиссию за транзакцию, которая добавляется к поощрительной стоимости блока, содержащего транзакцию. Как только заданное количество монет поступит в обращение, поощрение может полностью составлять комиссию за

транзакцию и быть полностью свободным от инфляции.

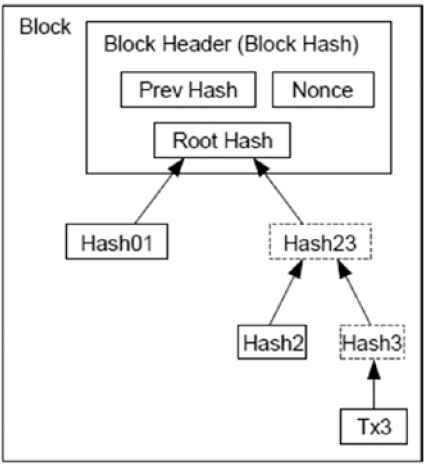
Стимул может помочь побудить узлы оставаться честными. Если жадный злоумышленник сможет собрать больше процессорной мощности, чем все честные узлы, ему придется выбирать между использованием ее для обмана людей путем кражи их платежей или использованием ее для создания новых монет. Он должен найти более выгодным играть по правилам, по таким правилам, которые дают ему больше новых монет, чем все остальные вместе взятые, чем подрывать систему и законность своего собственного богатства.

Восстановление места на диске

После того, как последняя транзакция в монете будет скрыта под достаточным количеством блоков, потраченные транзакции перед ней могут быть отброшены для экономии места на диске. Чтобы облегчить это, не нарушая хэш блока, транзакции хешируются в дереве Меркла [7][2][5], при этом в хеш блока включается только корень. Затем старые блоки можно уплотнить, обрубив ветки дерева. Внутренние хэши хранить не нужно.



Transactions Hashed in a Merkle Tree



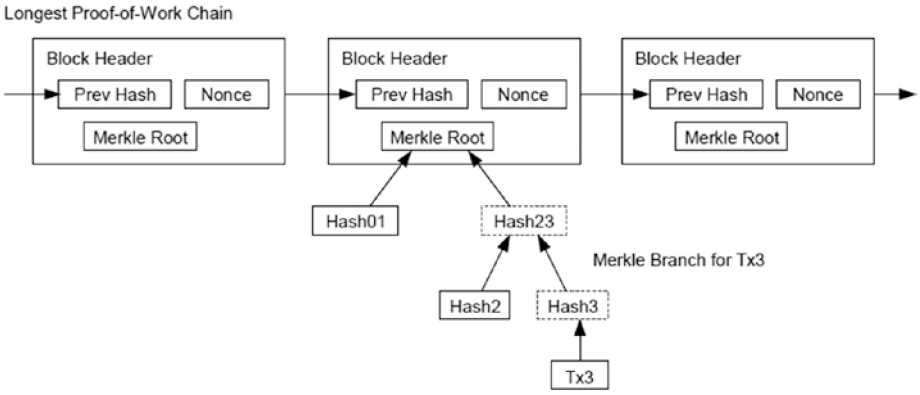
After Pruning Tx0-2 from the Block

Заголовок блока без транзакций будет иметь размер около 80 байт. Если предположить, что блоки генерируются каждые 10 минут, 80 байт * 6 * 24 * 365 = 4,2 Мб в год. Для компьютерных системам, которые обычно продаются с 2 Гб ОЗУ (по состоянию на 2008 год, а закон Мура предсказывает текущий рост на 1,2 Гб в год), хранение не должно быть проблемой, даже в том случае, если заголовки блоков должны храниться в памяти.

Упрощенная проверка платежа

Можно проверять платежи без запуска полного сетевого узла.

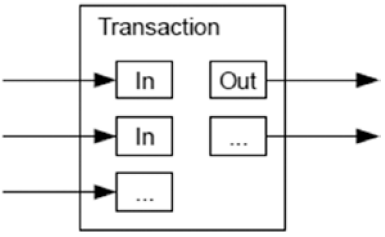
Пользователю нужно только сохранить копию заголовков блоков самой длинной цепочки проверки работоспособности, которую он может получить, постоянно запрашивая сетевые узлы, пока не убедится в том, что у него есть самая длинная цепочка, а также получить дерево Меркла, связывающее транзакцию с блоком, в котором оно имеет отметку времени. Пользователь не может сам проверить транзакцию, но, связав ее с местом в цепочке, он может увидеть, что сетевой узел принял транзакцию, а блоки, добавленные после нее, еще раз подтверждают, что сеть транзакцию приняла.



Таким образом, проверка надежна, пока сеть контролируется честными узлами, но более уязвима, если сеть перегружена злоумышленниками. В то время как сетевые узлы могут проверять транзакции для себя, упрощенный метод может быть скомпроментирован поддельными транзакциями злоумышленника до тех пор, пока злоумышленник сможет продолжать подавлять сеть. Одной из стратегий защиты от этого может быть прием предупреждений от сетевых узлов, когда они обнаруживают недопустимый блок, предлагая программному обеспечению пользователя загрузить полный блок и предупреждая транзакции для подтверждения несоответствия. Предприятия, которые получают частые платежи, вероятно, по-прежнему захотят запускать свои собственные узлы для более независимой безопасности и более быстрой проверки.

Объединение и разделение стоимости

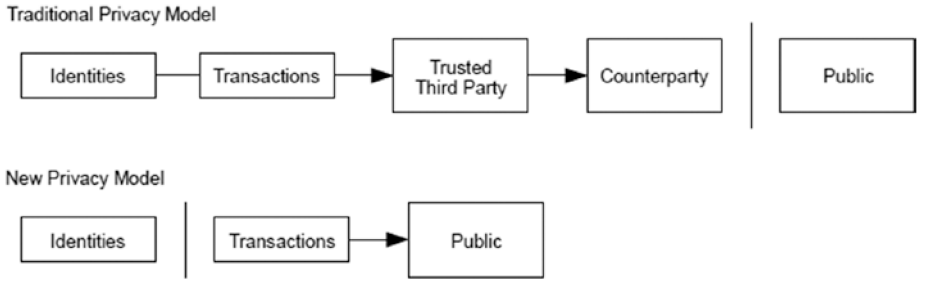
Хотя существует возможность обрабатывать монеты по отдельности, но было бы крайне неудобно производить отдельную транзакцию для каждого цента в переводе. Чтобы можно было разделить и объединить стоимость, транзакции содержат несколько входов и выходов. Обычно будет либо один вход из более крупной предыдущей транзакции, либо несколько входов, объединяющих меньшие суммы, и не более двух выходов: один для платежа и один для возврата сдачи, если таковая имеется, обратно отправителю.



Следует отметить, что разветвление, когда транзакция зависит от нескольких транзакций, а сами эти транзакции зависят от многих других, здесь не проблема. Никогда нет необходимости извлекать полную автономную копию истории транзакций.

Конфиденциальность

Традиционная банковская модель обеспечивает определенный уровень конфиденциальности, ограничивая доступ к информации вовлеченными сторонами и доверенной третьей стороной. Необходимость объявлять обо всех транзакциях публично исключает этот метод, но конфиденциальность все же можно поддерживать, прерывая поток информации в другом месте: сохраняя анонимность открытых ключей. Общественность может видеть, что кто-то отправляет кому-то некую сумму, но без информации, привязывающей транзакцию к кому-либо. Это похоже на уровень информации, публикуемой фондовыми биржами, где время и размер отдельных сделок, «лента», обнародуются, но без указания сторон.



В качестве дополнительного брандмауэра для каждой транзакции следует использовать новую пару ключей, чтобы предотвратить их привязку к общему владельцу. Некоторая привязка по-прежнему неизбежна для транзакций с несколькими входами, которые обязательно показывают, что их входы

принадлежат одному и тому же владельцу. Риск заключается в том, что если владелец ключа будет раскрыт, привязка может выявить другие транзакции, принадлежащие этому же владельцу.

Вычисления

Мы рассматриваем сценарий, когда злоумышленник пытается сгенерировать альтернативную цепочку быстрее, чем легитимную цепочку. Даже если это будет сделано, это не сделает систему открытой для произвольных изменений, таких как создание актива из воздуха или изъятие денег, которые никогда не принадлежали злоумышленнику. Узлы не примут недействительную транзакцию в качестве оплаты, а легитимные узлы никогда не примут блок, содержащий их. Злоумышленник может попытаться изменить только одну из своих транзакций, чтобы вернуть деньги, которые он недавно потратил.

Конкуренцию между легитимной цепью и цепью злоумышленника можно охарактеризовать как биномиальное случайное блуждание. Событием успеха является расширение честной цепочки на один блок, что увеличивает ее преимущество на +1, а событием неудачи является расширение цепочки атакующего на один блок, что уменьшает разрыв на -1.

Вероятность того, что злоумышленник наверстает упущенное, аналогична проблеме разорения игрока. Предположим, игрок с неограниченным кредитом начинает с проигрыша и потенциально играет бесконечное количество попыток, чтобы попытаться достичь безубыточности. Мы можем рассчитать вероятность того, что он когда-либо достигнет безубыточности или что злоумышленник когда-либо догонит легитимную цепочку, следующим образом [8]:

p = вероятность того, что легитимный узел найдет следующий блок

q = вероятность того, что злоумышленник найдет следующий

блок

q_z = вероятность того, что злоумышленник когда-либо догонит вас, начиная с z блоков позади

$$q_z = \begin{cases} 1 & if\ p \le q \\ (q/p)^z & if\ p > q \end{cases}$$

Учитывая наше предположение, что p>q, вероятность падает экспоненциально по мере увеличения количества блоков, которые злоумышленник должен догнать. С неблагоприятными для него шансами, если он не сделает удачный рывок вперед на ранней стадии, его шансы станут исчезающе малыми, поскольку он будет отставать все больше и больше.

Теперь мы рассмотрим, как долго получатель новой транзакции должен ждать, прежде чем он будет достаточно уверен в том, что отправитель не может изменить транзакцию. Мы предполагаем, что отправитель является злоумышленником, который хочет заставить получателя поверить в то, что он оплатил ему какое-то время назад, а затем отзывает платеж, чтобы вернуть деньги себе по прошествии некоторого времени. Когда это произойдет, получатель будет предупрежден, но отправитель надеется, что будет слишком поздно.

Получатель генерирует новую пару ключей и передает открытый ключ отправителю незадолго до подписания. Это не позволяет отправителю подготовить цепочку блоков заранее, постоянно работая над ней, пока ему не повезет, а затем в этот момент выполняя транзакцию. Как только транзакция отправлена, нечестный отправитель начинает тайно работать над параллельной цепочкой, содержащей альтернативную версию его транзакции.

Получатель ждет, пока транзакция не будет добавлена в блок и после нее не будут связаны z блоков. Он не знает точного прогресса, достигнутого злоумышленником, но если предположить, что честные блоки заняли среднее ожидаемое время

на блок, потенциальный прогресс злоумышленника будет представлять собой распределение Пуассона с ожидаемым значением:

$$\lambda = z\frac{q}{p}$$

Чтобы определить вероятность того, что атакующий все еще может догнать сейчас, мы умножаем распределение Пуассона для каждой величины прогресса, который он мог бы сделать, на вероятность того, что он сможет догнать с данной точки:

$$\sum_{k=0}^{\infty}\frac{\lambda^k e^{-\lambda}}{k!}\begin{cases}(q/p)^{(z-k)} & if\ k \le z\\ 1 & if\ k > z\end{cases}$$

Перестановка, с целью не допустить суммирования бесконечного хвоста распределения...

$$1-\sum_{k=0}^{z}\frac{\lambda^k e^{-\lambda}}{k!}\left(1-(q/p)^{(z-k)}\right)$$

Преобразование в код на C...

```
#include
double AttackerSuccessProbability(double q, int z)
{
    double p = 1.0 - q;
    double lambda = z * (q / p);
    double sum = 1.0;
    int i, k;
    for (k = 0; k <= z; k++)
    {
        double poisson = exp(-lambda);
        for (i = 1; i <= k; i++)
          poisson *= lambda / i;
```

```
            sum -= poisson * (1 - pow(q / p, z - k));
        }
        return sum;
}
```

Запустив код и получив некоторые результаты, мы можем увидеть, что вероятность экспоненциально падает с z.

```
q=0.1
z=0     P=1.0000000
z=1     P=0.2045873
z=2     P=0.0509779
z=3     P=0.0131722
z=4     P=0.0034552
z=5     P=0.0009137
z=6     P=0.0002428
z=7     P=0.0000647
z=8     P=0.0000173
z=9     P=0.0000046
z=10    P=0.0000012

q=0.3
z=0     P=1.0000000
z=5     P=0.1773523
z=10    P=0.0416605
z=15    P=0.0101008
z=20    P=0.0024804
z=25    P=0.0006132
z=30    P=0.0001522
z=35    P=0.0000379
z=40    P=0.0000095
z=45    P=0.0000024
z=50    P=0.0000006
```

Решение для P менее 0,1%...

P < 0.001
q=0.10 z=5
q=0.15 z=8
q=0.20 z=11
q=0.25 z=15
q=0.30 z=24
q=0.35 z=41
q=0.40 z=89
q=0.45 z=340

Заключение

Мы предложили систему для электронных транзакций, не полагаясь на доверие. Мы начали с обычной структуры монет, сделанных из цифровых подписей, которая обеспечивает строгий контроль над владением, но неполна без способа предотвращения двойной оплаты. Чтобы решить эту проблему, мы предложили одноранговую сеть, использующую доказательство работы для записи общедоступной истории транзакций, которую злоумышленнику быстро становится нецелесообразно изменять с вычислительной точки зрения, если легитимные узлы контролируют большую часть вычислительной мощности. Сеть надежна в своей неструктурированной простоте. Узлы работают одновременно с небольшой координацией. Их не нужно идентифицировать, поскольку сообщения не направляются в какое-либо конкретное место и должны быть только доставлены с максимальной эффективностью. Узлы могут покидать сеть и присоединяться к ней по своему желанию, принимая цепочку проверки работоспособности как доказательство того, что

произошло, пока они отсутствовали. Узлы голосуют вычислительной мощностью своего процессора, выражая свое согласие с легитимными блоками, работая над их расширением, и отвергая недействительные блоки и отказываясь работать над ними. Любые необходимые правила и стимулы могут быть применены с помощью этого механизма консенсуса.

Библиография

1. W. Dai, "b-money," http://www.weidai.com/bmoney.txt, 1998.
2. H. Massias, X.S. Avila, and J.-J. Quisquater, "Design of a secure timestamping service with minimal trust requirements," In *20th Symposium on Information Theory in the Benelux*, May 1999.
3. S. Haber, W.S. Stornetta, "How to time-stamp a digital document," In *Journal of Cryptology*, vol 3, no 2, pages 99-111, 1991.
4. D. Bayer, S. Haber, W.S. Stornetta, "Improving the efficiency and reliability of digital time-stamping," In *Sequences II: Methods in Communication, Security and Computer Science*, pages 329-334, 1993.
5. S. Haber, W.S. Stornetta, "Secure names for bit-strings," In *Proceedings of the 4th ACM Conference on Computer and Communications Security*, pages 28-35, April 1997.
6. Back, "Hashcash - a denial of service counter-measure," http://www.hashcash.org/papers/hashcash.pdf, 2002.
7. R.C. Merkle, "Protocols for public key cryptosystems," In *Proc. 1980 Symposium on Security and Privacy*, IEEE Computer Society, pages 122-133, April 1980.
8. W. Feller, "An introduction to probability theory and its applications," 1957.