Моше Задка
Марк Уильямс
Джулиан Берман
Том Мост



TWISTED

из первых рук

Событийное и асинхронное программирование на Python

ДМК ПРЕСС
издательство

УДК 004.438
ББК 32.973.22
315

Задка М., Уильямс М., Бенфилд К., Уорнер Б.,
Митчелл Д., Сэмюэл К., Тарди П.
315 Twisted из первых рук / пер. с анг. А. Н. Киселева. – М.: ДМК Пресс, 2020. – 338 с.: ил.

ISBN 978-5-97060-795-4

Эта книга посвящена Twisted – событийно-ориентированному сетевому фреймворку на Python, в котором можно создавать уникальные проекты. В первой части рассматриваются особенности Twisted; на практических примерах показано, как его архитектура способствует тестированию, решает общие проблемы надежности, отладки и упрощает выявление причинно-следственных связей, присущих событийно-ориентированному программированию. Детально описываются приемы асинхронного программирования, подчеркивается важность отложенного вызова функций и сопрограмм. На примере использования двух популярных приложений, treq и klein, демонстрируются сложности, возникающие при реализации веб-API с Twisted, и способы их преодоления.

Вторая часть книги посвящена конкретным проектам, использующим Twisted. В число примеров входят использование Twisted с Docker, применение Twisted в роли контейнера WSGI, организация обмена файлами и многое другое.

Читатель должен иметь некоторый опыт работы с Python и понимать основы контейнеров и протоколов. Знакомство с Twisted и с проектами, описанными в книге, не требуется.

УДК 004.438
ББК 32.973.22

Authorized Russian translation of the English edition of Expert Twisted ISBN 978-1-4842-3741-0 © 2019 Moshe Zadka, Mark Williams, Cory Benfield, Brian Warner, Dustin Mitchell, Kevin Samuel, Pierre Tardy.

This translation is published and sold by permission of Packt Publishing, which owns or controls all rights to publish and sell the same.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN 978-1-4842-3741-0 (анг.)
ISBN 978-5-97060-795-4 (рус.)

© 2019 Moshe Zadka, Mark Williams, Cory Benfield, Brian Warner, Dustin Mitchell, Kevin Samuel, Pierre Tardy
© Оформление, издание, перевод, ДМК Пресс, 2020

Содержание

Об авторах 12

Благодарности 14

Введение 15

От издательства 16

Часть I. ОСНОВЫ 17

Глава 1. Введение в событийно-ориентированное программирование с помощью Twisted 18

Примечание о версиях Python 19
Событийно-ориентированное программирование – что это? 19
Многократные события 20
Мультиплексирование и демультиплексирование 22
Мультиплексор select 23
  История, аналоги и назначение 23
  Сокеты и select 24
  События сокета – как, что и почему 25
  Обработка событий 26
  Цикл обработки событий с select 27
  Управляемые событиями клиенты и серверы 29
Неблокирующий ввод/вывод 31
  Знаем, когда нужно остановиться 31
  Отслеживание состояния 32
  Наличие информации о состоянии усложняет программы 35
Управление сложностью с помощью транспортов и протоколов 36
  Реакторы: работа с транспортом 37
Транспорты: работа с протоколами 37
  Игра в пинг-понг с протоколами и транспортами 38
  Клиенты и серверы со своими реализациями протоколов и транспортов 42
  Реакторы Twisted и протоколы с транспортами 43
Преимущества событийно-ориентированного программирования 44
Twisted и реальный мир 46
События и время 50
  Повторение событий с LoopingCall 53
Интерфейсы событий в zope.interface 55

Управление потоком в событийно-ориентированных программах 57
Управление потоком в Twisted с помощью производителей и потребителей 58
Активные производители 59
Потребители 61
Пассивные производители 64
Итоги 64

Глава 2. Введение в асинхронное программирование с Twisted 66

Обработчики событий и их композиция 66
Что такое асинхронное программирование? 69
Заполнители для будущих значений 70
Асинхронная обработка исключений 72
Введение в Twisted Deferred 76
Обычные обработчики 76
Ошибки и их обработчики 77
Композиция экземпляров Deferred 80
Генераторы и inlineCallbacks 83
yield 83
send 84
throw 86
Асинхронное программирование с inlineCallbacks 87
Сопрограммы в Python 89
Сопрограммы с выражением yield from 90
Сопрограммы async и await 91
Ожидание завершения экземпляров Deferred 96
Преобразование сопрограмм в Deferred с помощью ensureDeferred 97
Мультиплексирование объектов Deferred 98
Тестирование объектов Deferred 102
Итоги 105

Глава 3. Создание приложений с библиотеками treq и Klein 107

Насколько важную роль играют эти библиотеки? 107
Агрегирование каналов 108
Введение в treq 109
Введение в Klein 112
Klein и Deferred 113
Механизм шаблонов Plating в Klein 115
Первая версия агрегатора каналов 117
Разработка через тестирование с использованием Klein и treq 123
Выполнение тестов на устанавливаемом проекте 123
Тестирование Klein с помощью StubTreq 126
Тестирование treq с помощью Klein 133

Журналирование с использованием twisted.logger ... 136
Запуск приложений Twisted с помощью twist ... 141
Итоги ... 144

Часть II. ПРОЕКТЫ ... 146

Глава 4. Twisted в Docker ... 147

Введение в Docker ... 147
Контейнеры ... 147
Образы контейнеров ... 148
runc и containerd ... 149
Клиент ... 149
Реестр ... 150
Сборка ... 150
Многоступенчатая сборка ... 151
Python в Docker ... 153
Варианты развертывания ... 153
В виртуальном окружении ... 157
В формате Pex ... 159
Варианты сборки ... 160
Один большой образ ... 160
Копирование пакетов wheel между этапами ... 161
Копирование окружения между этапами ... 161
Копирование файлов Pex между этапами ... 161
Автоматизация с использованием Dockerpy ... 161
Twisted в Docker ... 162
ENTRYPOINT и PID 1 ... 162
Пользовательские плагины ... 162
NColony ... 162
Итоги ... 165

Глава 5. Использование Twisted в роли сервера WSGI ... 166

Введение в WSGI ... 166
PEP ... 167
Простой пример ... 168
Базовая реализация ... 170
Пример WebOb ... 172
Пример Pyramid ... 173
Начало ... 174
Сервер WSGI ... 174
Поиск кода ... 177
Путь по умолчанию ... 177
PYTHONPATH ... 177

setup.py ... 177
Почему Twisted? ... 178
Промышленная эксплуатация и разработка ... 178
TLS ... 179
Индикация имени сервера ... 180
Статические файлы ... 182
Модель ресурсов ... 182
Чисто статические ресурсы ... 183
Комбинирование статических файлов с WSGI ... 185
Встроенное планирование задач ... 186
Каналы управления ... 189
Стратегии параллельного выполнения ... 191
Балансировка нагрузки ... 191
Открытие сокета в режиме совместного использования ... 192
Другие варианты ... 195
Динамическая конфигурация ... 195
Приложение Pyramid с поддержкой A/B-тестирования ... 195
Плагин для поддержки AMP ... 197
Управляющая программа ... 200
Итоги ... 201

Глава 6. Tahoe-LAFS: децентрализованная файловая система ... 202

Как работает Tahoe-LAFS ... 203
Архитектура системы ... 206
Как система Tahoe-LAFS использует Twisted ... 208
Часто встречающиеся проблемы ... 208
Инструменты поддержки выполнения в режиме демона ... 209
Внутренние интерфейсы FileNode ... 210
Интеграция интерфейсных протоколов ... 211
Веб-интерфейс ... 212
Типы файлов, Content-Type, /named/ ... 214
Сохранение на диск ... 215
Заголовки Range ... 215
Преобразование ошибок на возвращающей стороне ... 216
Отображение элементов пользовательского интерфейса: шаблоны Nevow ... 217
Интерфейс FTP ... 218
Интерфейс SFTP ... 223
Обратная несовместимость Twisted API ... 223
Итоги ... 226
Ссылки ... 226

Глава 7. Magic Wormhole 227
Как это выглядит 228
Как это работает 229
Сетевые протоколы, задержки передачи, совместимость клиентов 231
Сетевые протоколы и совместимость клиентов 231
Архитектура сервера 234
База данных 235
Транзитный клиент: отменяемые отложенные операции 235
Сервер транзитной ретрансляции 238
Архитектура клиента 239
Отложенные вычисления и конечные автоматы, одноразовый наблюдатель 240
Одноразовые наблюдатели 243
Promise/Future и Deferred 244
Отсроченные вызовы, синхронное тестирование 247
Асинхронное тестирование с объектами Deferred 248
Синхронное тестирование с объектами Deferred 249
Синхронное тестирование и отсроченный вызов 252
Итоги 254
Ссылки 254

Глава 8. Передача данных в браузерах и микросервисах с использованием WebSocket 255
Нужен ли протокол WebSocket? 255
WebSocket и Twisted 256
WebSocket, из Python в Python 258
WebSocket, из Python в JavaScript 261
Более мощная поддержка WebSocket в WAMP 263
Итоги 269

Глава 9. Создание приложений с asyncio и Twisted 271
Основные понятия 271
Механизм обещаний 272
Циклы событий 273
Рекомендации 274
Пример: прокси с aiohttp и treq 277
Итоги 280

Глава 10. Buildbot и Twisted 282
История появления Buildbot 282
Эволюция асинхронного выполнения кода на Python в Buildbot 283
Миграция синхронных API 286
Этапы асинхронной сборки 287

Код Buildbot ... 287
Асинхронные утилиты ... 288
«Дребезг» ... 288
Асинхронные службы ... 288
Кеш LRU ... 291
Отложенный вызов функций ... 291
Взаимодействие с синхронным кодом ... 292
SQLAlchemy ... 292
requests ... 293
Docker ... 295
Конкурентный доступ к общим ресурсам ... 296
yield как барьер конкуренции ... 296
Функции из пула потоков не должны изменять общее состояние ... 297
Блокировки Deferred ... 298
Тестирование ... 298
Имитации ... 300
Итоги ... 300

Глава 11. Twisted и HTTP/2 ... 301

Введение ... 301
Цели и задачи ... 303
Бесшовная интеграция ... 303
Оптимизация поведения по умолчанию ... 304
Разделение задач и повторное использование кода ... 305
Проблемы реализации ... 306
Что такое соединение? Ценность стандартных интерфейсов ... 306
Мультиплексирование и приоритеты ... 309
Противодавление ... 315
Противодавление в Twisted ... 317
Противодавление в HTTP/2 ... 319
Текущее положение дел и возможность расширения в будущем ... 321
Итоги ... 322

Глава 12. Twisted и Django Channels ... 323

Введение ... 323
Основные компоненты Channels ... 325
Брокеры сообщений и очереди ... 325
Распределенные многоуровневые системы в Twisted ... 327
Текущее положение дел и возможность расширения в будущем ... 328
Итоги ... 329

Предметный указатель ... 330

Глава 1

Введение в событийно-ориентированное программирование с помощью Twisted

Twisted – это мощная, хорошо протестированная, развитая параллельная сетевая библиотека и фреймворк. Как мы в этой книге увидим далее, многие организации и отдельные люди при создании своих проектов эффективно использовали этот фреймворк на протяжении более чем десятка лет.

В то же время Twisted большой, сложный и старый. Его лексикон изобилует странными названиями, такими как «реактор», «протокол», «конечная точка» и «отложенный». Эти понятия описывают философию и архитектуру, сбивающую с толку как новичков, так и людей с многолетним опытом работы с Python.

Сетевые приложения, создаваемые с помощью Twisted, основываются на двух основных принципах программирования: *событийного программирования* и *асинхронного программирования*. Рост популярности языка программирования JavaScript и включение в стандартную библиотеку Python пакета asyncio привели к тому, что оба принципа стали основой фреймворка. Но ни один из принципов не занимает настолько доминирующего положения в программировании на Python, чтобы простого знания языка было достаточно для их освоения. Эти сложные темы доступны, пожалуй, только программистам со средним или высоким уровнем подготовки.

В этой и следующей главах представлена мотивация, лежащая в основе событийно-управляемого и асинхронного программирования, а затем показано, как Twisted использует эти принципы. Здесь закладывается основа для последующих глав, в которых рассматриваются реальные программы Twisted.

Сначала мы познакомимся с идеей событийно-ориентированного программирования без привязки к Twisted. Затем, получив представление о том, что определяет событийно-управляемое программирование, мы познакомимся с программными абстракциями в Twisted, помогающими разработчикам писать четкие и эффективные событийные программы. Попутно мы будем делать остановки, чтобы познакомиться с некоторыми уникальными частями этих абстракций, например *интерфейсами*, и исследуем документацию с их описанием на веб-сайте Twisted.

К концу этой главы вы освоите терминологию Twisted: протоколы, транспорты, реакторы, потребители и производители. Данные понятия образуют основу событийно-ориентированного программирования с Twisted. Знание этой основы необходимо для разработки полезного программного обеспечения с Twisted.

Примечание о версиях Python

Twisted поддерживает Python 2 и 3, поэтому все примеры кода в этой главе могут работать как с Python 2, так и с Python 3. Python 3 – это будущее, но одной из сильных сторон Twisted является его богатая история реализации протоколов; по этой причине важно, чтобы вы умели писать код, способный выполняться под управлением Python 2, даже если прежде вы никогда не писали его.

Событийно-ориентированное программирование – что это?

Событие – это то, что заставляет управляемую событиями программу выполнить действие. Это широкое определение позволяет рассматривать многие программы как управляемые событиями. Рассмотрим, например, простую программу, которая в зависимости от ввода пользователя печатает либо `Hello`, либо `World`:

```
import sys
line = sys.stdin.readline().strip()
if line == "h":
    print("Hello")
else:
    print("World")
```

Появление строки в потоке стандартного ввода является событием. Наша программа приостанавливается на операторе `sys.stdin.readline()`, который просит операционную систему разрешить пользователю ввести законченную строку. Пока ввод не будет получен, программа дальше двигаться не будет. Когда операционная система прочитает очередной ввод и внутренние компоненты Python определят, что строка завершена, оператор `sys.stdin.`

readline() возобновит работу программы, вернув ей полученные данные. Возобновление работы – это событие, толкающее нашу программу вперед. Поэтому даже такую простую программу можно представить как управляемую событиями.

Многократные события

Программа, обрабатывающая единственное событие и завершающая работу, не получает никаких преимуществ от событийно-ориентированного подхода. Программы, в которых одновременно может происходить несколько событий, имеют более естественную организацию для их обработки. Примером такой программы может быть графический интерфейс пользователя: в любой момент пользователь может нажать кнопку, выбрать пункт из меню, прокрутить текст и т. д.

Вот версия нашей предыдущей программы с графическим интерфейсом Tkinter:

```
from six.moves import tkinter
from six.moves.tkinter import scrolledtext
class Application(tkinter.Frame):
    def __init__ (self, root):
        super(Application,self). __init__ (root)
        self.pack()
        self.helloButton = tkinter.Button(self,
                                    text="Say Hello",
                                    command=self.sayHello)
        self.worldButton = tkinter.Button(self,
                                     text="Say World",
                                     command=self.sayWorld)
        self.output = scrolledtext.ScrolledText(master=self)
        self.helloButton.pack(side="top")
        self.worldButton.pack(side="top")
        self.output.pack(side="top")
    def outputLine(self, text):
        self.output.insert(tkinter.INSERT, text+ '\n')
    def sayHello(self):
        self.outputLine("Hello")
    def sayWorld(self):
        self.outputLine("World")

Application(tkinter.Tk()).mainloop()
```

Эта версия нашей программы представляет пользователю две кнопки, каждая из которых может генерировать независимое событие (click) и отличается от предыдущей программы тем, что в предыдущей программе sys.stdin.readline может генерировать только одно событие: «готовность строки».

С этими событиями мы связали *обработчики событий*, которые вызываются нажатием любой кнопки. Кнопки Tkinter имеют свойство command со ссылкой

на обработчик, и вызывают этот обработчик при нажатии. Так, когда нажимается кнопка **Say Hello**, она генерирует событие, вынуждающее программу вызвать функцию `Application.sayHello`, которая, в свою очередь, выводит слово Hello в текстовое окно с прокруткой. То же происходит при нажатии кнопки **Say World**, которая вызывает функцию `Application.sayWorld` (см. рис. 1.1).



Рис. 1.1 ❖ Приложение Tkinter GUI
после нескольких нажатий кнопок **Say Hello** и **Say World**

Метод `tkinter.Frame.mainloop`, унаследованный нашим классом `Application`, ждет, пока связанная с ним кнопка сгенерирует событие, и запускает соответствующий обработчик. После выполнения каждого обработчика `tkinter.Frame.mainloop` переходит к ожиданию новых событий. Цикл ожидания событий и их передачи в соответствующие обработчики типичен для управляемых событиями программ и известен как *цикл событий*.

Вот основные понятия, лежащие в основе событийно-ориентированного программирования.

1. *События* показывают, что произошло то, на что должна реагировать программа. В обоих наших примерах события естественно соответствуют нажатиям кнопок, но, как мы увидим, они могут представлять все, что заставляет нашу программу выполнять какое-либо действие.
2. *Обработчики событий* определяют реакцию программы на события. Иногда обработчик события имеет вид простой инструкции, как вызов `sys.stdin.readline` в нашем примере. Но чаще всего он представлен функцией или методом, как в нашем примере `tkinter`.

3. *Цикл событий* ждет появления события и вызывает соответствующий обработчик. Не все управляемые событиями программы имеют цикл событий; в нашем первом примере `sys.stdin.readline` цикла обработки событий нет, потому что здесь происходит только одно событие. Однако большинство программ напоминают наш пример `tkinter` тем, что перед окончательным завершением обрабатывают много событий. Такие программы используют цикл событий.

Мультиплексирование и демультиплексирование

Способ ожидания появления событий в цикле существенно влияет на написание управляемых событиями программ, поэтому рассмотрим его более подробно. Вернемся к нашему примеру `tkinter` с двумя кнопками. Цикл событий внутри `mainloop` должен ждать, пока пользователь не нажмет одну из двух кнопок.

Простая реализация цикла может выглядеть следующим образом:

```
def mainloop(self):
    while self.running:
        ready = [button for button in self.buttons if button.hasEvent()]
        if ready:
            self.dispatchButtonEventHandlers(ready)
```

Цикл `mainloop` в ожидании нового события постоянно *опрашивает* каждую кнопку и запускает обработчики только для имеющих готовое событие. Когда событий нет, программа никаких действий не выполняет, так как никаких действий, требующих ответа, предпринято не было. В течение периодов бездействия управляемая событиями программа свое выполнение должна приостанавливать.

Цикл `while` в нашем примере `mainloop` приостанавливает работу программы до тех пор, пока не будет нажата одна из кнопок и не понадобится вызвать функцию `sayHello` или `sayWorld`. Если пользователь не сможет щелкать мышью со сверхъестественной скоростью, большая часть времени цикла будет тратиться на проверку кнопок, которые не были нажаты. Это называется *активным ожиданием*.

Активное ожидание, подобное этому, приостанавливает общее выполнение программы до тех пор, пока один из ее источников событий не сообщит о том, что событие произошло. Поэтому активного ожидания достаточно для приостановки цикла событий.

Внутренний генератор списков, управляющий активным ожиданием, задает важный вопрос: что-нибудь произошло? Ответ помещается в переменную `ready` и имеет вид списка со всеми кнопками, которые были нажаты. Истинность переменной `ready` определяет ответ на вопрос: если переменная `ready` ничего не содержит (имеет ложное значение), значит, кнопки не нажи-

мались и ничего не произошло. Если переменная имеет истинное значение, следовательно, событие произошло и по меньшей мере одна из кнопок была нажата.

Генератор списков, формирующий значение для `ready`, объединяет множество событий в одном месте. Этот процесс известен как *мультиплексирование*. Обратный процесс разделения списка на отдельные события называется *демультиплексированием*. Генератор списка объединяет (мультиплексирует) все наши кнопки в переменной `ready`, тогда как метод `dispatchButtonEventHandlers` демультиплексирует (перенаправляет сигнал с одного из информационных входов на один из информационных выходов) их, вызывая обработчик каждого события отдельно.

Теперь мы можем уточнить наше понимание циклов событий, точно описав, как происходит ожидание событий.

- *Цикл событий* ожидает появления событий, мультиплексируя их источники в один список. Если получился непустой список, цикл событий демультиплексирует его и для каждого события вызывает соответствующий обработчик.

Наш мультиплексор `mainloop` тратит бóльшую часть своего времени на опрос кнопок, которые не были нажаты. Но не все мультиплексоры настолько неэффективны. В `tkinter.Frame.mainloop` используется аналогичный мультиплексор, который опрашивает все виджеты, если операционная система не предлагает более эффективного способа. Мультиплексор `mainloop`, чтобы повысить свою эффективность, использует тот факт, что компьютеры могут проверять виджеты GUI быстрее, чем с ними может взаимодействовать человек, и вставляет вызов `sleep`, приостанавливающий программу на несколько миллисекунд. Это позволяет программе часть времени в цикле активного ожидания вообще не выполнять никаких операций и за счет небольшой задержки экономить процессорное время и электроэнергию.

Хоть Twisted может интегрироваться с графическими пользовательскими интерфейсами и фактически имеет специальную поддержку `tkinter`, в его основе лежит сетевой движок. В сети основными источниками событий являются *сокеты*, а не кнопки, и операционные системы предлагают эффективные механизмы для мультиплексирования событий сокетов. Цикл событий в Twisted использует эти механизмы. Чтобы понять подход Twisted к событийному программированию, нужно разобраться, как взаимодействуют сокеты с этими механизмами мультиплексирования.

Мультиплексор select

История, аналоги и назначение

Мультиплексор `select` поддерживается большинством операционных систем. Свое имя «select» (выбор) этот мультиплексор получил из-за своей способно-

сти выбирать из всего списка сокетов только те, у которых есть готовые к обработке события.

Мультиплексор `select` появился в 1983 году, когда возможности компьютеров были гораздо скромнее. Как следствие, он имеет не лучшую эффективность, особенно при мультиплексировании большого количества сокетов. Каждое семейство операционных систем предоставляет свой, более эффективный мультиплексор, такой как `kqueue` в BSD и `epoll` в Linux, но между собой они не взаимодействуют. К счастью, эти мультиплексоры имеют схожий принцип работы, и мы можем обобщить их поведение. Используем для иллюстрации мультиплексор `select`.

Сокеты и select

В следующем коде отсутствует обработка ошибок, поэтому он будет терпеть неудачу во многих пограничных случаях, встречающихся на практике. **Этот код приводится исключительно в демонстрационных целях. Не используйте его в реальных приложениях.** Twisted стремится правильно обрабатывать ошибки и крайние случаи, поэтому имеет довольно сложную реализацию.

Теперь, предупредив вас об опасностях, начнем интерактивный сеанс Python и создадим сокеты для мультиплексирования с использованием `select`:

```
>>> import socket
>>> listener = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>>> listener.bind(('127.0.0.1', 0))
>>> listener.listen(1)
>>> client = socket.create_connection(listener.getsockname())
>>> server, _ = listener.accept()
```

Полное объяснение сокетов выходит за рамки этой книги. Фактически мы надеемся, что после обсуждения деталей вы предпочтете Twisted! Однако приведенный выше код содержит больше фундаментальных понятий, чем несущественных деталей:

1) `listener` (прослушиватель) – сокет, предназначенный для приема входящих соединений. Это сокет интернета (`socket.AF_INET`) и TCP (`socket.SOCK_STREAM`), доступный клиентам через внутренний локальный сетевой интерфейс (который обычно имеет адрес 127.0.0.1) и порт, случайно выбираемый операционной системой (0). Данный прослушиватель выполняет настройку входящего соединения и ставит его в очередь до тех пор, пока оно не будет прочитано (`listen(1)`);
2) `client` (клиент) – сокет исходящего соединения. Функция `socket.create_connection` принимает кортеж (хост, порт), представляющий прослушивающий сокет, к которому требуется подключиться, и возвращает подключенный сокет. Поскольку наш сокет `listener` находится в том же процессе, мы можем получить его хост и порт с помощью `listener.getsockname()`;

3) `server` (сервер) – входящее соединение на сервере. После подключения клиента к нашему хосту и порту необходимо принять соединение из очереди `listener`. `listener.accept` возвращает кортеж (сокет, адрес). Так как нам нужен только сокет, адрес мы отбрасываем. Реальная программа может записать адрес в журнал или использовать его для отслеживания метрик подключения. Очередь прослушивателя с длиной 1, которую мы создали вызовом метода `listen`, хранит этот сокет, пока мы не вызовем `accept`, что позволит функции `create_connection` вернуть управление.

`client` (клиент) и `server` (сервер) – это два конца одного TCP-соединения. Установленное TCP-соединение не различает клиента и сервер; клиентский сокет обладает теми же правами на чтение, запись или закрытие соединения, что и серверный:

```
>>> data = b"xyz"
>>> client.sendall(data)
>>> server.recv(1024) == data
True
>>> server.sendall(data)
>>> client.recv(1024) == data
True
```

События сокета – как, что и почему

Внутри операционной системы для каждого сокета TCP поддерживаются буферы чтения и записи для учета ненадежности сети, а также клиентов и серверов с различной скоростью записи-чтения. Если сервер временно не доступен и не может получать данные `b"xyz"`, переданные в `client.sendall`, то они останутся в буфере записи до тех пор, пока сервер снова не станет активным. Точно так же, если клиент слишком занят и не смог вовремя вызвать `client.recv` и получить данные `b"xyz"`, отправленные вызовом `server.sendall`, они будут храниться в буфере чтения клиента, пока тот не получит их. Число, которое мы передаем `recv`, представляет максимальное количество данных, которые мы готовы удалить из буфера чтения. Если объем данных в буфере чтения меньше указанной величины, как в данном примере, `recv` удалит из буфера и возвратит *все* данные.

Двунаправленность наших сокетов подразумевает два возможных события:

1) *событие готовности к чтению* означает, что сокет получил какие-то данные и их можно прочитать. Серверный сокет генерирует это событие, когда в его приемный буфер попадают какие-то данные, то есть вызов `recv` сразу после события готовности к чтению немедленно вернет эти данные. Если данных не окажется в буфере, произойдет разъединение. Коме того, прослушивающий сокет генерирует это событие, когда мы можем принять новое соединение;
2) *событие доступности к записи* означает, что в буфере записи сокета есть свободное место. Это тонкий момент: пока сервер успевает обрабаты-

вать данные быстрее, чем мы добавляем их в буфер отправки клиентского сокета, этот буфер остается доступным для записи.

Интерфейс `select` отражает эти возможные события и принимает до четырех аргументов:

1) список сокетов для мониторинга *событий готовности к чтению*;
2) список сокетов для мониторинга *событий доступности для записи*;
3) список сокетов для мониторинга «исключительных событий». В наших примерах исключительных событий не предполагается, поэтому мы всегда будем передавать пустой список;
4) необязательный *тайм-аут*. Это количество секунд, в течение которых `select` будет ждать, когда хотя бы один из подконтрольных сокетов сгенерирует событие. Если опустить этот аргумент, `select` будет ждать вечно.

Мы можем спросить у `select`, какие события сгенерировали наши сокеты:

```
>>> import select
>>> maybeReadable = [listener, client, server]
>>> maybeWritable = [client, server]
>>> readable, writable, _ = select.select(maybeReadable, maybeWritable, [], 0)
>>> readable
[]
>>> writable == maybeWritable and writable == [client, server]
True
```

Мы потребовали от `select` не ждать новых событий, передав тайм-аут 0. Как объяснялось выше, сокеты `client` и `server` могут быть доступны как для чтения, так и для записи, а сокет `listener` может только принимать входящие соединения, то есть может генерировать лишь события готовности к чтению.

Если опустить аргумент с тайм-аутом, `select` приостановит нашу программу до тех пор, пока один из контролируемых сокетов не станет доступным для чтения или записи. Приостановка выполнения производится так же, как в примере мультиплексирования с циклом активного ожидания, опрашивающим все кнопки в нашей упрощенной реализации `mainloop`.

Вызов `select` мультиплексирует сокеты более эффективно, чем активное ожидание, потому что операционная система возобновит нашу программу только в том случае, когда будет сгенерировано хотя бы одно событие. Внутри ядра цикл событий, в отличие от нашего `select`, ожидает событий от сетевого оборудования и затем передает их нашему приложению.

Обработка событий

`select` возвращает кортеж с тремя списками в том же порядке, в котором расположены его аргументы. Обход каждого списка *демультиплексирует* возвращаемое значение `select`. Ни один из наших сокетов не сгенерировал события готовности к чтению, хотя мы записывали данные и в `client`, и в `server`. Наши предыдущие вызовы `recv` очистили их буферы чтения, и к `listener` никаких

новых подключений не происходило с тех пор, как было принято соединение с клиентом и создан сокет server. Однако в буферах отправки client и server есть свободное место, поэтому они оба сгенерировали событие доступности для записи.

Отправка данных с client на server приводит к тому, что server генерирует событие готовности к чтению, поэтому select помещает его в список readable:

```
>>> client.sendall(b'xyz')
>>> readable, writable, _ = select.select(maybeReadable, maybeWritable, [], 0)
>>> readable == [server]
True
```

Список доступных для записи сокетов, что интересно, снова содержит наши сокеты client и server:

```
>>> writable == maybeWritable and writable == [client, server]
True
```

Если еще раз вызвать select, сокет server снова будет готов к чтению, и оба сокета – client и server – опять будут доступны для записи. Причина проста: пока данные остаются в буфере чтения сокета, для него непрерывно будет генерироваться событие готовности к чтению, а пока в передающем буфере сокета остается место, для него будет генерироваться событие доступности для записи. Мы можем это подтвердить, прочитав данные, полученные сокетом server, и снова вызвав select:

```
>>> server.recv(1024) == b'xyz'
True
>>> readable, writable, _ = select.select(maybeReadable, maybeWritable,
[], 0)
>>> readable
[]
>>> writable == maybeWritable and writable == [client, server]
True
```

Очистка буфера чтения сервера привела к тому, что он перестал генерировать событие готовности к чтению, но оба сокета – client и server – продолжают генерировать событие доступности для записи, потому что в их буферах записи еще есть место.

Цикл обработки событий с select

Теперь мы знаем, как select мультиплексирует сокеты:

1) различные сокеты генерируют готовности к чтению/записи, чтобы сообщить управляемой событиями программе, что та может принять входящие данные или соединения или записать исходящие данные;
2) select мультиплексирует сокеты, проверяя их готовность к чтению/записи, и приостанавливает программу, пока на появится хотя бы одно событие или не истечет тайм-аут;

3) сокеты продолжают генерировать события готовности к чтению/записи до тех пор, пока не изменятся обстоятельства, приведшие к этим событиям: сокет с данными в буфере чтения продолжает генерировать событие готовности к чтению, пока буфер не опустеет; прослушивающий сокет продолжает генерировать событие готовности к чтению, пока все входящие соединения не будут приняты; а сокет, в буфере отправки которого еще есть место, будет генерировать события доступности для записи, пока буфер не заполнится.

Опираясь на эти знания, мы можем набросать цикл событий с `select`:

```
import select

class Reactor(object):
    def __init__ (self):
        self._readers = {}
        self._writers = {}
    def addReader(self, readable, handler):
        self._readers[readable] = handler
    def addWriter(self, writable, handler):
        self._writers[writable] = handler
    def removeReader(self, readable):
        self._readers.pop(readable,None)
    def removeWriter(self, writable):
        self._writers.pop(writable,None)
    def run(self):
        while self._readers or self._writers:
            r, w, _ = select.select(list(self._readers), list
            (self._writers), [])
            for readable in r:
                self._readers[readable](self, readable)
            for writable in w:
                if writable in self._writers:
                    self._writers[writable](self, writable)
```

Мы назвали наш цикл событий *реактором*, потому что он реагирует на события сокета. Мы можем попросить `Reactor` запускать наши обработчики событий готовности к чтению, вызывая его метод `addReader`, и обработчики событий доступности для записи, вызывая `addWriter`. Обработчики событий принимают два аргумента: сам реактор и сокет, который сгенерировал событие.

Цикл внутри метода `run` мультиплексирует сокеты с помощью `select`, а затем демультиплексирует результат на сокеты, готовые к чтению и доступные для записи. Сначала вызываются обработчики для сокетов, готовых к чтению. Далее, прежде чем запустить обработчик события доступности для записи, цикл событий проверяет, зарегистрирован ли по-прежнему сокет в словаре `_writers`. Это важно, потому что закрытие соединения генерирует событие готовности к чтению, и запущенный непосредственно перед этим обработчик чтения может удалить закрытый сокет из словарей `_readers` и `_writers`. К моменту запуска обработчика события доступности для записи закрытый сокет будет удален из словаря `_writers`.

Управляемые событиями клиенты и серверы

Этого простого цикла событий достаточно для реализации клиента, который постоянно записывает данные на сервер. Начнем с обработчиков событий:

```
def accept(reactor, listener):
    server, _ = listener.accept()
    reactor.addReader(server, read)
def read(reactor, sock):
    data = sock.recv(1024)
    if data:
        print("Server received", len(data),"bytes.")
    else:
        sock.close()
        print("Server closed.")
        reactor.removeReader(sock)

DATA=[b"*", b"*"]
def write(reactor, sock):
    sock.sendall(b"".join(DATA))
    print("Client wrote", len(DATA)," bytes.")
    DATA.extend(DATA)
```

Функция `accept` обрабатывает событие готовности к чтению сокета `listening`, принимая входящее соединение и передавая реактору соответствующий сокет для проверки наличия в нем событий готовности к чтению. Они обрабатываются функцией `read`.

Функция `read` обрабатывает событие готовности к чтению, пытаясь получить фиксированный объем данных из приемного буфера сокета. Она выводит длину принятого сообщения – напомню, что число, переданное в `recv`, представляет *верхний предел* количества возвращаемых байтов. Если в сокете, сгенерировавшем событие готовности к чтению, данные отсутствуют, значит, другая сторона соединения закрыла свой сокет. Поэтому функция `read` в ответ закрывает свой сокет и удаляет его из набора сокетов, контролируемых реактором на появление события готовности к чтению. Закрытие сокета освобождает ресурсы операционной системы, а удаление его из реактора гарантирует, что мультиплексор `select` не будет пытаться контролировать неактивный сокет.

Функция `write` записывает последовательность звездочек (*) в сокет, сгенерировавший событие доступности для записи. После каждой успешной записи объем данных удваивается. Это имитирует поведение реальных сетевых приложений, которые не всегда записывают в соединение одинаковый объем данных. Рассмотрим веб-браузер: некоторые исходящие запросы содержат небольшое количество данных из формы, введенных пользователем, в то время как другие могут выгружать огромные файлы.

Обратите внимание, что это функции уровня модуля, а не методы нашего класса `Reactor`. Вместо этого они ассоциируются с реактором путем их регистрации в качестве обработчиков событий доступности для чтения/записи, потому что TCP-сокеты – это только один вид сокетов и события в других сокетах

может потребоваться обрабатывать иначе. Однако сама функция `select` обрабатывает все сокеты одинаково, поэтому логика вызова обработчиков событий для сокетов в возвращаемых списках должна инкапсулироваться классом `Reactor`. Позже мы рассмотрим, насколько для программ, управляемых событиями, важны инкапсуляция и интерфейсы, которые она подразумевает.

Теперь можно создать сокеты прослушивателя `listener` и клиента `client` и позволить циклу событий управлять приемом соединений и передачей данных между клиентским и серверным сокетами.

```
import socket
listener = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
listener.bind(('127.0.0.1',0))
listener.listen(1)
client = socket.create_connection(listener.getsockname())

loop = Reactor()
loop.addWriter(client, write)
loop.addReader(listener, accept)
loop.run()
```

После запуска эта программа в какой-то момент терпит неудачу:

```
Client wrote 2 bytes.
Server received 2 bytes.
Client wrote 4 bytes.
Server received 4 bytes.
Client wrote 8 bytes.
Server received 8 bytes.
...
Client wrote 524288 bytes.
Server received 1024 bytes.
Client wrote 1048576 bytes.
Server received 1024 bytes.
^CTraceback (most recent call last):
  File "example.py", line 53, in <module>
    loop.run()
  File "example.py", line 25, in run
    writeHandler(self, writable)
  File "example.py", line 33, in write
    sock.sendall(b"".join(DATA))
KeyboardInterrupt
```

Сначала все хорошо: данные передаются из клиентского сокета в серверный. Это поведение соответствует логике обработчиков событий `accept`, `read` и `write`. Как и ожидалось, сначала клиент посылает серверу два байта `b'*'`, который, в свою очередь, получает эти два байта.

Одновременная работа клиента и сервера демонстрирует эффективность событийно-ориентированного программирования. Подобно нашему графическому приложению, реагировавшему на события от двух разных кнопок, этот небольшой сетевой сервер способен откликаться на события от клиента или сервера, что позволило объединить их в одном процессе. Мультиплексирую-

щие способности select обеспечивают единую точку в цикле событий, где программа может реагировать на любое из них.

Но потом возникла проблема: после определенного количества повторений программа зависает, и ее приходится прерывать с помощью клавиатуры. Ключ к этому находится в выводе трассировки стека нашей программы; спустя какое-то время клиент застревает, пытаясь переслать большой объем данных серверу, как видно по трассировке стека после прерывания KeyboardInterrupt, ведущей прямо к вызову sock.sendall в нашем обработчике записи (write).

Сервер не поспевает за клиентом, в результате чего большую часть времени буфер отправки клиентского сокета остается заполненным до отказа. По умолчанию, если в буфере отправки нет места, функция sendall приостанавливает (*блокирует*) программу. Если бы sendall *не* блокировала программу и наш цикл событий был бы мог выполняться как обычно, сокет не был бы доступен для записи, и блокирующий вызов sendall не выполнялся бы. Однако мы не можем с уверенностью сказать, сколько данных нужно передать функции sendall, чтобы она заполнила буфер отправки, не заблокировав при этом программу, обработчик записи выполнился бы до конца, и функция select предотвратила бы дальнейшие попытки записи, пока в буфере не освободится место. Природа сетей такова, что о подобной проблеме мы узнаем только после того, как она возникнет.

Все события, которые мы до сих пор рассматривали, заставляют программу выполнять какие-то действия. Но мы еще не уделили внимания событиям, заставляющим программу выполняемые действия *прекратить*. Нам нужен новый вид события.

Неблокирующий ввод/вывод

Знаем, когда нужно остановиться

По умолчанию сокеты блокируют программу, начавшую операцию, которая не может быть завершена, пока на удаленном конце не произойдет определенное действие. Для предотвращения блокировки мы можем заставить сокет выдавать событие, попросив операционную систему сделать его *неблокирующим*.

Давайте вернемся к интерактивному сеансу Python и снова построим соединение между клиентским (client) и серверным (server) сокетами. На этот раз мы сделаем сокет client неблокирующим и попытаемся записать в него бесконечный поток данных.

```
>>> import socket
>>> listener = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>>> listener.bind(( '127.0.0.1',0))
>>> listener.listen(1)
>>> client=socket.create_connection(listener.getsockname())
>>> server, _ = listener.accept()
```

```
>>> client.setblocking(False)
>>> while True: client.sendall(b"*"*1024)
...
Traceback (most recent call last):
  File"<stdin>", line1, in <module>
BlockingIOError: [Errno11] Resource temporarily unavailable
```

Мы снова заполнили буфер отправки клиента, но вместо того, чтобы приостановить процесс, `sendall` вызвала исключение. Тип исключения зависит от версии Python. В Python 3 это исключение `BlockingIOError`, а в Python 2 это общее исключение `socket.error`. В обеих версиях Python значение атрибута `errno` исключения будет установлено в `errno.EAGAIN`:

```
>>> import errno, socket
>>> try:
...     while True: client.sendall(b"*"*1024)
... except socket.error as e:
...     print(e.errno == errno.EAGAIN)
True
```

Исключение – это сгенерированное операционной системой событие, указывающее, что мы должны *прекратить* запись. Этого почти достаточно, чтобы исправить проблему в наших клиенте и сервере.

Отслеживание состояния

Однако для обработки этого исключения требуется ответить на новый вопрос: сколько данных, которые мы пытались записать, попали в буфер отправки сокета? Не ответив на этот вопрос, мы не сможем узнать, какие данные действительно были отправлены. Не зная этого, мы не сможем правильно писать программы с неблокирующими сокетами. Например, веб-браузер обязан отслеживать количество и объем выгруженных файлов. Иначе содержимое может быть повреждено в пути.

`client.sendall`, прежде чем сгенерировать исключение с ошибкой `EAGAIN`, может поместить в буфер отправки любое количество байтов. Мы должны отказаться от метода `sendall` и вместо него использовать метод `send`, который возвращает объем данных, записанных в буфер отправки сокета. Мы можем продемонстрировать это с помощью сокета `server`:

```
>>> server.setblocking(False)
>>> try:
...     while True: print(server.send(b"*" * 1024))
... except socket.error as e:
...     print("Terminated with EAGAIN:", e.errno == errno.EAGAIN)
1024
1024
...
1024
952
Terminated with EAGAIN:True
```

Мы переводим сокет server в неблокирующий режим, чтобы по заполнении буфера отправки он генерировал событие EAGAIN. Затем цикл while вызывает server.send. Вызовы, возвращающие 1024, записали все предоставленные байты в буфер отправки сокета. В конце концов, буфер записи сокета заполняется, и исключение с ошибкой EAGAIN завершает цикл. Однако последний успешный вызов send перед завершением цикла вернул 952 – он просто отбросил последние 72 байта. Это известно как *короткая запись*. То же происходит и с блокирующими сокетами! Когда в буфере отправки не останется свободного места, sendall не генерирует исключение, а запускает цикл, который проверяет значение, возвращаемое вызовом send, и повторяет эти вызовы до тех пор, пока не будут отправлены все данные.

В нашем случае размер буфера отправки сокета не был кратным 1024, поэтому мы не смогли отправить круглое число байтов в последнем вызове send, прежде чем возникла ошибка EAGAIN. Однако в реальном мире размер буфера отправки сокета изменяется в зависимости от условий в сети, и приложения отправляют по соединениям различные объемы данных. Программы, использующие неблокирующий ввод/вывод, например как наш гипотетический веббраузер, регулярно сталкиваются с короткими записями, подобными этим.

Используя значение, возвращаемое функцией send, можно гарантировать запись всех данных в соединение. Мы можем добавить свой буфер для данных, которые нужно записать. Каждый раз, когда select создаст для сокета событие доступности для записи, мы попытаемся отправить данные, находящиеся в этот момент в буфере. Если вызов send завершится без ошибки EAGAIN, мы запомним результат и удалим это количество байтов из начала нашего буфера, потому что send записывает данные в буфер отправки, выбирая их из начала полученной последовательности. Если send сгенерирует ошибку EAGAIN, указывающую, что буфер отправки полностью заполнен и не может вместить больше данных, мы оставим содержимое буфера без изменений. Так будет продолжаться, пока наш собственный буфер не опустеет, и когда это произойдет, мы будем знать, что все наши данные помещены в буфер отправки сокета. После этого операционная система должна отправить все накопленные данные на приемный конец соединения.

Теперь исправим наш простой пример клиент-сервер, разделив функцию write на два элемента – функцию, которая инициирует запись данных, и объект, управляющий буфером и вызывающий метод send:

```
import errno
import socket

class BuffersWrites(object):
    def __init__ (self, dataToWrite, onCompletion):
        self._buffer = dataToWrite
        self._onCompletion = onCompletion
    def bufferingWrite(self, reactor, sock):
        if self._buffer:
```

```
            try:
                written = sock.send(self._buffer)
            except socket.error as e:
                if e.errno != errno.EAGAIN:
                    raise
                return
            else:
                print("Wrote", written,"bytes")
                self._buffer = self._buffer[written:]
        if not self._buffer:
                reactor.removeWriter(sock)
                self._onCompletion(reactor, sock)

DATA=[b"*", b"*"]
def write(reactor, sock):
    writer = BuffersWrites(b"".join(DATA), onCompletion=write)
    reactor.addWriter(sock, writer.bufferingWrite)
    print("Client buffering", len(DATA),"bytes to write.")
    DATA.extend(DATA)
```

Первый аргумент метода инициализации `BuffersWrites` – это последовательность байтов для отправки, которая будет играть роль буфера. А второй аргумент, `onCompletion`, является вызываемым объектом. Как следует из названия, `onCompletion` (по завершении) будет вызван после записи всех данных в буфер отправки сокета.

Сигнатура метода `bufferingWrite` отвечает требованиям функции `Reactor.addWriter` к обработчикам событий доступности для записи. Он пытается отправить данные из буфера в сокет и запоминает число отправленных байтов. Если `send` вызывает исключение `EAGAIN`, `bufferingWrite` перехватывает это исключение и просто выходит; если вызов `send` сгенерирует какое-то другое исключение, оно продолжит распространение вверх по стеку вызовов. В обоих случаях `self._buffer` остается неизменным.

Если отправка выполнена успешно, записанное число байтов удаляется из начала буфера `self._buffer` и `buffureWrite` завершается. Например, если вызов `send` запишет только `952` байта из `1024`, `self_buffer` будет содержать последние 72 байта.

Наконец, если буфер опустел, значит, все запрошенные данные были записаны, и для экземпляра `BuffersWrites` не осталось работы. Он обращается к реактору, чтобы прекратить мониторинг своего сокета на наличие событий доступности для записи, а затем вызывает `onCompletion`, чтобы сообщить, что все данные записаны. Обратите внимание, что эта проверка выполняется в операторе `if`, который не зависит от первого оператора `if self._buffer`. Предыдущий код мог выполниться и очистить буфер; если бы заключительный код находился в блоке `else` оператора `if self._buffer`, он не был бы выполнен до следующего раза, когда реактор обнаружит, что сокет доступен для записи. Чтобы упростить управление ресурсами, мы вынесли проверку в отдельный оператор `if`.

Функция `write` выглядит аналогично предыдущей версии, за исключением