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.

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

Содержание

Об авторах

Марк Уильямс (Mark Williams) работает в Twisted. В eBay и PayPal Марк Уильямс работал над высокопроизводительными веб-службами Python (более миллиарда запросов в день!), над обеспечением безопасности приложений и информации, а также переносом корпоративных Python-библиотек на Python.

Кори Бенфилд (Cory Benfield) – разработчик Python с открытым исходным кодом, активно участвует в сообществе Python HTTP. Входит в число основных разработчиков проектов Requests и urllib3 и является ведущим сопровождающим проекта Hyper – коллекции инструментов поддержки HTTP и HTTP/2 для Python, а также помогает в разработке Python Cryptographic Authority для PyOpenSSL.

Брайан Уорнер (Brian Warner) – инженер по безопасности и разработчик программного обеспечения, работавший в Mozilla на Firefox Sync, Add-On SDK и Persona. Один из основателей проекта Tahoe-LAFS – распределенной и защищенной файловой системы, разрабатывает средства безопасного хранения и связи.

Моше Задка (Moshe Zadka) с 1995 года является участником сообщества открытого исходного кода, свой первый вклад в Python внес в 1998 году, один из основателей открытого проекта Twisted. Любит рассказывать о Twisted и Python и выступать на конференциях. Регулярно ведет блоги.

Дастин Митчелл (Dustin Mitchell) внес свой вклад в Buildbot. Является членом команды TaskCluster в Mozilla. Также работает в командах Release Engineering, Release Operations и Infrastructure.

Кевин Сэмюэл (Kevin Samuel) начал заниматься разработкой и преподаванием еще во времена, когда появилась версия Python 2.4. Работал в Восточной Европе, Северной Америке, Азии и Западной Африке. Тесно сотрудничает с командой Crossbar.io и является активным членом французского сообщества Python.

О технических рецензентах

Пьер Тарди (Pierre Tardy) – специалист по непрерывной интеграции в Renault Software Labs, в настоящее время является ведущим коммитером в Buildbot.

Джулиан Берман (Julian Berman) – разработчик программного обеспечения с открытым исходным кодом из Нью-Йорка. Автор библиотеки jsonschema для Python, периодически вносит вклад в экосистему Twisted, активный участник сообщества Python.

Шон Шоджи (Shawn Shojaie) живет в районе Калифорнийского залива, где работает инженером-программистом. Работал в Intel, NetApp. Сейчас работает в SimpleLegal, где создает веб-приложения для юридических фирм. В будние дни занимается разработкой с использованием Django и PostgreSQL, а в выходные вносит вклад в проекты с открытым исходным кодом, такие как django-pylint, и время от времени пишет технические статьи. Больше можно узнать на сайте shawnshojaie.com.

Том Мост (Tom Most) – Twisted-коммитер с десятилетним опытом разработки веб-служб, клиентских библиотек и приложений командной строки с использованием Twisted. Инженер-программист в телекоммуникационной отрасли. Сопровождает Afkak, клиента Twisted Kafka. Его адрес в интернете – freecog.net, а связаться можно по адресу twm@freecog.net.

Введение

Twisted недавно отпраздновал свой шестнадцатый день рождения. За время своего существования он превратился в мощную библиотеку. За этот период на основе Twisted было создано несколько интересных приложений и многие из нас узнали много нового о том, как правильно использовать Twisted, как думать о сетевом коде и как создавать программы, основанные на событиях.

После ознакомления с вводными материалами, которые есть на сайте Twisted, часто можно услышать: «И что делать дальше? Как я могу узнать больше о Twisted?» На этот вопрос мы обычно отвечали встречным вопросом: «А что вы хотите сделать с Twisted?» В этой книге мы покажем некоторые интересные приемы использования Twisted.

Авторы данной книги использовали Twisted для разных целей и усвоили разные уроки. Мы рады представить все эти уроки, чтобы сообщество их знало и могло воспользоваться.

Вперед!

Часть I

ОСНОВЫ

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).

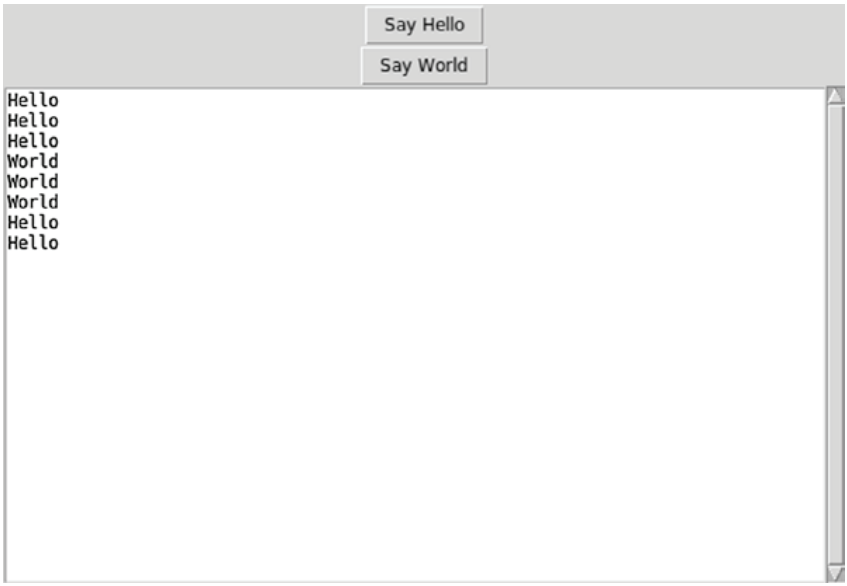

Рис. 1.1 ❖ Приложение Tkinter GUI
после нескольких нажатий кнопок **Say Hello** и **Say World**

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

того, что теперь она делегирует отправку данных методу `bufferingWrite` объекта `BuffersWrites`. Обратите особое внимание на то, что `write` передает *себя* в `BuffersWrites` в роли вызываемого объекта `onCompletion`. Такая *косвенная рекурсия* создает тот же эффект цикла, что и в предыдущей версии. `write` никогда не вызывает себя напрямую, а передает себя объекту, который, в свою очередь, вызывается *реактором*. Эта косвенность позволяет продолжать данную последовательность без переполнения стека вызовов.

С этими изменениями наша программа больше не блокируется. Но теперь она терпит неудачу по другой причине: в какой-то момент объем данных становится слишком большим и не помещается в доступной памяти компьютера! Вот, например, что получилось на компьютере автора:

```
Client buffering 2 bytes to write.
Wrote 2 bytes
Client buffering 4 bytes to write.
Server received 2 bytes.
Wrote 4 bytes
...
Client buffering 2097152 bytes to write.
Server received 1024 bytes.
Wrote 1439354 bytes
Server received 1024 bytes.
Server received 1024 bytes.
....
Wrote 657798 bytes
Server received 1024 bytes.
Server received 1024 bytes.
....
Client buffering 268435456 bytes to write.
Traceback (most recent call last):
  File "example.py", line 76, in <module>
    loop.run()
  File "example.py", line 23, in run
    writeHandler(self, writable)
  File "example.py", line 57, in bufferingWrite
    self._onCompletion(reactor, sock)
  File "example.py", line 64, in write
    DATA.extend(DATA)
MemoryError
```

Наличие информации о состоянии усложняет программы

Несмотря на проблему, связанную с переполнением памяти компьютера, мы успешно справились с созданием управляемой событиями программы, которая для управления записью данных в сокеты использует неблокирующий ввод/вывод. Однако код получился запутанным: косвенный вызов `write` через `BuffersWrites` и реактор мешает понять логический поток исходящих данных, и очевидно, что реализация чего-то более сложного, чем отправка последовательности звездочек, потребует расширения классов и интерфейсов. Напри-

мер, как обработать исключение `MemoryError`? Наш подход непригоден для реальных приложений.

Управление сложностью с помощью транспортов и протоколов

Программирование с неблокирующим вводом/выводом, несомненно, имеет ряд сложностей. Вот что пишет об этом администратор UNIX W. Ричард Стивенс в первом томе своей основополагающей серии *Unix Network Programming*[1]:

> Но будут ли оправданы затраченные усилия при написании приложения, использующего неблокируемый ввод-вывод, с учетом усложнения итогового кода? Нет, ответим мы.
>
> (*UNIX. Разработка сетевых приложений, с. 478*)

Сложность нашего кода, похоже, доказывает справедливость утверждения Стивенса. Правильные абстракции, однако, могут загерметизировать сложность в управляемом интерфейсе. В нашем примере уже есть многократно используемый код: любая новая единица кода, выполняющая запись в сокет, должна использовать основную логику `BuffersWrites`. Мы *загерметизировали* сложность операции записи в неблокирующий сокет. Основываясь на этом понимании, мы можем выделить две концептуальные области:

1) *транспорты*: `BuffersWrites` управляет процессом записи выходных данных в неблокирующий сокет *независимо от их содержимого*. Он может пересылать фотографии, музыку – любые данные, какие только можно себе представить. Главное условие – эти данные перед передачей должны быть преобразованы в байты. `BuffersWrites` – это *транспорт*, или, другими словами, *средство транспортировки* байтов. Транспорты герметизируют (инкапсулируют) процесс чтения данных из сокета, а также прием новых соединений. Транспорт не только инициирует действия в нашей программе, но и получает от программы результаты собственных действий;
2) *протоколы*: программа, которую мы привели в качестве примера, генерирует данные с помощью простого алгоритма и подсчитывает получаемые данные. Более сложные программы могут создавать веб-страницы или преобразовывать в текст голосовые телефонные сообщения. Пока эти программы могут принимать и отдавать байты, они могут согласованно работать с тем, что мы назвали транспортом. Они также могут управлять поведением своего транспорта. Например, при получении недопустимых данных закрывать активное соединение. В области телекоммуникаций такие правила, определяющие порядок передачи дан-

[1] *У. Р. Стивенс, Б. Феннер, Э. М. Рудофф*. UNIX. Разработка сетевых приложений. Т. 1. 3-е изд. СПб.: Питер, 2007. ISBN: 5-318-00535-7. – *Прим. перев.*

ных, описываются в виде *протокола*. Протокол, таким образом, *определяет, как генерировать и обрабатывать входные и выходные данные*. То есть протокол инкапсулирует *эффект* нашей программы.

Реакторы: работа с транспортом

Мы изменим наш реактор `Reactor` и приспособим его для работы с транспортом:

```
import select
class Reactor(object):
    def __init__ (self):
        self._readers = set()
        self._writers = set()
    def addReader(self, transport):
        self._readers.add(transport)
    def addWriter(self, transport):
        self._writers.add(transport)
    def removeReader(self, readable):
        self._readers.discard(readable)
    def removeWriter(self, writable):
        self._writers.discard(writable)
    def run(self):
        while self._readers or self._writers:
            r, w, _ = select.select(self._readers,self._writers, [])
            for readable in r:
                readable.doRead()
            for writable in w:
                if writable in self._writers:
                    writable.doWrite()
```

Если раньше наши обработчики событий готовности к чтению/записи были функциями, то теперь они являются методами объектов-транспортов: `doRead` и `doWrite`. Кроме того, теперь реактор мультиплексирует не сокеты, а транспорты. С точки зрения реактора интерфейс транспорта включает:

1) `doRead`;
2) `doWrite`;
3) что-то, что делает состояние транспорта видимым для `select`: метод `fileno()`, возвращающий число, распознаваемое `select` как ссылка на сокет.

Транспорты: работа с протоколами

Теперь вернемся к функциям `read` и `write` и рассмотрим реализацию протокола. На функцию `read` возлагается две обязанности:

1) подсчет количества байтов, полученных из сокета;
2) выполнение некоторых действий в ответ на закрытие соединения.

Функция `write` имеет единственную обязанность: помещать данные в очередь для записи.

Из вышеописанного мы можем сделать набросок первого проекта интерфейса протокола:

```
class Protocol(object):
    def makeConnection(self, transport):
        ...
    def dataReceived(self, data):
        ...
    def connectionLost(self, exceptionOrNone):
        ...
```

Обязанности `read` мы разделили на два метода: `dataReceived` и `connectionLost`. Сигнатура первого говорит сама за себя, а вот второй требует некоторых пояснений: он получает один аргумент – объект исключения, если соединение было закрыто из-за исключения (например, `ECONNRESET`), или `None`, если оно было закрыто не из-за ошибки (например, когда `read` не получила никаких данных). Обратите внимание, что в нашем интерфейсе протокола отсутствует метод `write`. Это связано с тем, что запись данных, которая предполагает транспортировку байтов, попадает в зону ответственности транспорта. То есть экземпляр `Protocol` должен иметь доступ к транспорту, представляющему сетевое подключение и имеющему метод `write`. Связь между этими двумя объектами создается с помощью метода `makeConnection`, принимающего транспорт в качестве аргумента.

Почему бы не передать аргумент с транспортом методу инициализации протокола? Использование отдельного метода может показаться излишеством, но такой подход дает больше гибкости; например, представьте, как этот метод позволит нам добавить кеширование в протокол. Более того, так как транспорт вызывает методы протокола `dataReceived` и `connectionLost`, он тоже должен иметь доступ к протоколу. Если бы обоим классам `Transport` и `Protocol` требовалось передать в метод инициализации друг друга, возникла бы циклическая связь, мешающая создать оба экземпляра. Мы решили разорвать эту циклическую связь и добавить в `Protocol` отдельный метод, принимающий транспорт.

Игра в пинг-понг с протоколами и транспортами

Этого достаточно, чтобы написать более сложный протокол, использующий новый интерфейс. В предыдущем примере клиент просто посылал серверу все увеличивающуюся последовательности байтов; мы можем сделать так, чтобы после достижения необязательного максимума получатель закрывал соединение.

```
class PingPongProtocol(object):
    def __init__ (self, identity, maximum=None):
        self._identity = identity
        self._received = 0
        self._maximum = maximum
    def makeConnection(self, transport):
```

```
        self.transport = transport
        self.transport.write(b'*')
    def dataReceived(self, data):
        self._received += len(data)
        if self._maximum is not None and self._received >= self._maximum:
            print(self._identity,"is closing the connection")
            self.transport.loseConnection()
        else:
            self.transport.write(b'*')
            print(self._identity,"wrote a byte")
    def connectionLost(self, exceptionOrNone):
        print(self._identity,"lost the connection:", exceptionOrNone)
```

Метод инициализации принимает строку `identity`, идентифицирующую экземпляр протокола, и необязательный максимальный объем данных, получив который, следует закрыть соединение. `makeConnection` связывает `PingPongProtocol` с его транспортом и начинает обмен, отправляя один байт. `dataReceived` записывает объем полученных данных. Если он превышает необязательный максимум, `dataReceived` сообщает транспорту о потере соединения, что эквивалентно отключению. Иначе `dataReceived` продолжает обмен, отправляя обратно один байт. Наконец, после закрытия соединения протоколом `connectionLost` печатает сообщение.

`PingPongProtocol` описывает набор моделей поведения, сложность которых значительно превышает наши предыдущие попытки создания неблокирующего приложения клиент-сервер. В то же время его реализация отражает предшествующее ему прозаическое описание, не увязая в подробностях неблокирующего ввода/вывода. Мы смогли увеличить сложность нашего приложения и одновременно уменьшить сложность уникального управления вводом/выводом. Позже мы вернемся к изучению этих последствий, но достаточно сказать, что сужение нашего внимания позволяет устранять сложности в конкретных областях нашей программы.

Мы не можем использовать `PingPongProtocol`, пока не напишем транспорт. Создадим первый проект интерфейса транспорта:

```
class Transport(object):
    def __init__ (self, sock, protocol):
        ...
    def doRead(self):
        ...
    def doWrite(self):
        ...
    def fileno(self):
        ...
    def write(self):
        ...
    def loseConnection(self):
        ...
```

В первом аргументе методу инициализации передается сокет, который обертывает транспорт. Это обеспечивает инкапсуляцию сокетов в экземпляре `Transport`, с которыми в данное время работает реактор. Во втором аргументе передается протокол, чьи методы `dataReceived` и `connectionLost` будут вызываться при получении новых данных и закрытии соединения соответственно. Методы `doRead` и `doWrite` соответствуют описанному выше интерфейсу транспорта со стороны реактора. Частью этого интерфейса является новый метод `fileno`. Объект с правильно реализованным методом `fileno` можно передать в `select`. Наш метод `fileno` будет делегировать свои вызовы сокету, что сделает транспорты неотличимыми от сокетов с точки зрения `select`.

Метод `write` предоставляет интерфейс, который наш протокол использует для отправки исходящих данных. Мы на стороне протокола также добавили новый метод `loseConnection`, который инициирует закрытие сокета и представляет активную сторону закрытия соединения для пассивного метода `connectionLost`.

Мы можем реализовать этот интерфейс, встроив `BuffersWrites` и обработку сокетов в функцию `read`:

```
import errno

class Transport(object):
    def __init__ (self, reactor, sock, protocol):
        self._reactor = reactor
        self._socket = sock
        self._protocol = protocol
        self._buffer = b""
        self._onCompletion = lambda:None
    def doWrite(self):
        if self._buffer:
            try:
                written = self._socket.send(self._buffer)
            except socket.error as e:
                if e.errno != errno.EAGAIN:
                    self._tearDown(e)
                return
            else:
                print("Wrote", written,"bytes")
                self._buffer = self._buffer[written:]
        if not self._buffer:
            self._reactor.removeWriter(self)
            self._onCompletion()
    def doRead(self):
        data=self._socket.recv(1024)
        if data:
            self._protocol.dataReceived(data)
        else:
            self._tearDown(None)
    def fileno(self):
        return self._socket.fileno()
```

```
    def write(self, data):
        self._buffer += data
        self._reactor.addWriter(self)
        self.doWrite()
    def loseConnection(self):
        if self._buffer:
            def complete():
                self.tearDown(None)
            self._onCompletion = complete
        else:
            self._tearDown(None)
    def _tearDown(self, exceptionOrNone):
        self._reactor.removeWriter(self)
        self._reactor.removeReader(self)
        self._socket.close()
        self._protocol.connectionLost(exceptionOrNone)
    def activate(self):
        self._socket.setblocking(False)
        self._protocol.makeConnection(self)
        self._reactor.addReader(self)
        self._reactor.addWriter(self)
```

`doRead` и `doWrite` выполняют те же манипуляции с сокетами, которые в предыдущих примерах производили функции `read` и `write` и объект `BuffersWrites`. `doRead` также передает все полученные данные в метод протокола `dataReceived` или, получив пустую последовательность байтов, вызывает метод `connectionLost`. Наконец, `fileno` завершает требуемый реактору интерфейс, делая `Transports` пригодным для передачи в вызов `select`.

Метод `write`, как и раньше, буферизует запись, но вместо делегирования процесса записи отдельному классу он для сброса в сокет данных вызывает соседний метод `doWrite`. Если буфер пуст, вызов `loseConnection` разрывает соединение:

1) удаляя транспорт из реактора;
2) закрывая базовый сокет, чтобы освободить ресурсы сокета и вернуть их обратно в операционную систему;
3) передавая `None` в вызов метода `connectionLost` протокола, чтобы указать, что соединение было разорвано методом пассивного закрытия.

Если буфер содержит данные для записи, `loseConnection` переопределяет обработчик `_onCompletion` и подставляет замыкание, которое разрывает соединение, действуя так же, как было описано выше. Как и в случае с `BuffersWrites`, `Transport._onCompletion` вызывается только тогда, когда все байты в нашем буфере записи будут сброшены в базовый сокет. `loseConnection` так использует `_onCompletion`, что базовое соединение гарантированно остается открытым до тех пор, пока все данные не будут записаны. Обработчик `_onCompletion` по умолчанию устанавливается методом инициализации класса `Transport` как лямбда без эффекта. Это гарантирует, что многократные вызовы `write` смогут повторно

использовать исходное соединение. Реализации `write` и `lateConnection` совместно осуществляют транспортный интерфейс, требуемый протоколом.

Наконец, `activate` активирует транспорт:

1) подготавливая обернутый сокет к неблокирующему вводу/выводу;
2) передавая экземпляр `Transport` в свой протокол вызовом `Protocol.makeConnection`;
3) и наконец, регистрируя транспорт в реакторе.

На этом `Transport` завершает инкапсуляцию начала жизненного цикла своего сокета, конец которого уже инкапсулирован в `loseConnection`.

Если класс `Protocol` помог расширить сферу нашего внимания и добавить в приложение новые особенности поведения за счет определения `PingPongProtocol`, то `Transport` сузил ее, сконцентрировав на жизненном цикле ввода/вывода сокетов. Реактор – наш цикл обработки событий – определяет и отправляет события, связанные с сокетами, а протокол содержит обработчики этих событий. `Transport` выполняет посреднические функции, преобразуя события из сокетов в вызовы методов протокола и обеспечивая *последовательный* вызов этих методов, например он гарантирует вызов метода `makeConnection` протокола в начале жизненного цикла и вызов `loseConnection` – в конце. Это еще одно улучшение по сравнению с нашим примером клиента и сервера; мы сконцентрировали поток управления сокетами внутри транспорта `Transport`, не распыляя его по независимым функциям и объектам.

Клиенты и серверы со своими реализациями протоколов и транспортов

Показать обобщенность `Transport` можно, определив подтип `Listener`, который принимает входящие соединения и связывает их с уникальным экземпляром `PingPongProtocol`:

```
class Listener(Transport):
    def activate(self):
        self._reactor.addReader(self)

    def doRead(self):
        server, _ = self._socket.accept()
        protocol = PingPongProtocol("Server")
        Transport(self._reactor, server, protocol).activate()
```

Сокет, принимающий запросы на соединение, не генерирует событий доступности для записи, поэтому мы переопределяем метод `activate` и регистрируем транспорт только для чтения. Наш обработчик событий чтения `doRead` должен получить новое соединение с клиентом и протокол, а затем связать их вместе в активированном экземпляре транспорта.

Вот этап настройки для примера клиента и сервера, управляемых протоколом и транспортом:

```
listenerSock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
listenerSock.bind(('127.0.0.1',0))
```

```
listenerSock.listen(1)
clientSock = socket.create_connection(listenerSock.getsockname())

loop = Reactor()
Listener(loop, listenerSock, None).activate()
Transport(loop, clientSock, PingPongProtocol("Client", maximum=100)).
    activate()
loop.run()
```

Они будут обмениваться одиночными байтами, пока клиент не получит максимум 100 байт, после чего он закроет соединение:

```
Server wrote a byte
Client wrote a byte
Wrote 1 bytes
Server wrote a byte
Wrote 1 bytes
Client wrote a byte
Wrote 1 bytes
Server wrote a byte
Wrote 1 bytes
Client wrote a byte
Wrote 1 bytes
Server wrote a byte
Server wrote a byte
Client is closing the connection
Client lost the connection: None
Server lost the connection: None
```

Реакторы Twisted и протоколы с транспортами

Мы прошли долгий путь: начав с `select`, мы дошли до набора интерфейсов, инкапсулирующих цикл обработки событий, и обработчиков, которые четко разделяют обязанности. Нашей программой управляет экземпляр `Reactor`, а транспорты `Transport` отправляют события, порождаемые сокетами, в обработчики на уровне приложения, объявленные в протоколах `Protocol`.

Наши реакторы, транспорты и протоколы – это всего лишь пробные реализации. Например, вызов `socket.create_connection` блокирует приложение, и мы не исследовали никаких других неблокирующих альтернатив. Фактически базовое разрешение имен с помощью DNS, которое производится в `create_connection`, тоже может заблокировать приложение!

Однако концептуально они вполне готовы к серьезному использованию. Реакторы, транспорты и протоколы составляют основу событийно-ориентированной архитектуры Twisted. Как мы уже видели, их архитектура, в свою очередь, основана на мультиплексировании и неблокирующем выполнении ввода/вывода, что позволяет Twisted работать более эффективно.

Но прежде чем приступить к изучению самого фреймворка Twisted, рассмотрим наши примеры с высоты птичьего полета и оценим сильные и слабые стороны событийно-ориентированного программирования.